(19) JAPAN PATENT OFFICE

PATENT ABSTRACTS OF JAPAN

(11) Publication number: **11212808 A**

(43) Date of publication of application: **06.08.99**

(51) Int. Cl **G06F 9/46**
G06F 12/00
G06F 12/02

(21) Application number: **10260427**

(22) Date of filing: **14.09.98**

(30) Priority: **21.11.97 JP 09321766**

(71) Applicant: **OMRON CORP**

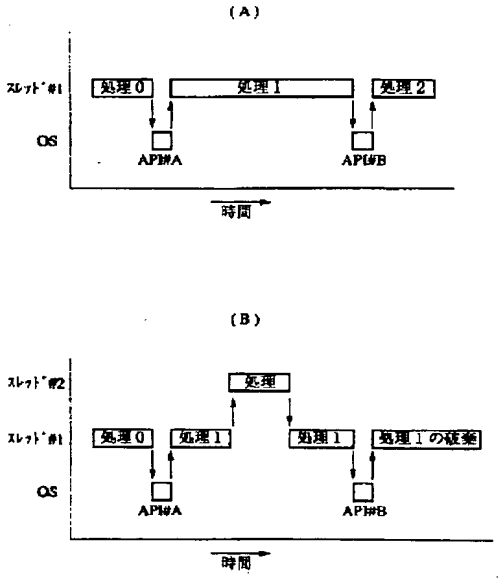
(72) Inventor: **HIRONO MITSUAKI**
INUI KAZUYUKI
KONAKA YOSHIHARU
KURIBAYASHI HIROSHI

(54) PROGRAM CONTROLLER, AND DEVICE AND METHOD FOR MEMORY ALLOCATION

(57) Abstract:

PROBLEM TO BE SOLVED: To secure real-time processing by guaranteeing the exclusiveness of a process without locking a system, to eliminates the need of compaction even by a mark & sweep method, to improve memory use efficiency, to reduce fragments in consideration of life for each object, and to reduce a waste of CPU power.

SOLUTION: An application program interface API#A which makes a request to start context switch generation presence/absence detection before the start of a process 1 is issued and an application program interface API#B which makes a request to end the context switch generation absence/presence detection at the end of the process 1 is issued. It is decided from the return value of this API#B whether or not context switching has been done halfway in the process 1 and when so, the process 1 is discarded.



(19)日本国特許庁（ＪＰ）　(12)特　許　公　報（Ｂ２）　(11)特許番号
特許第3027845号
(P3027845)

(45)発行日　平成12年４月４日(2000.4.4)　(24)登録日　平成12年２月４日(2000.2.4)

(51)Int.Cl.[7]	識別記号	FI	
G06F 9/46	340	G06F 9/46	340B
12/00	591	12/00	591
12/02	510	12/02	510A

請求項の数５(全 42 頁)

(21)出願番号　特願平10－260427
(22)出願日　平成10年９月14日(1998.9.14)
(65)公開番号　特開平11－212808
(43)公開日　平成11年８月６日(1999.8.6)
審査請求日　平成11年２月15日(1999.2.15)
(31)優先権主張番号　特願平9－321766
(32)優先日　平成９年11月21日(1997.11.21)
(33)優先権主張国　日本（ＪＰ）

(73)特許権者　000002945
オムロン株式会社
京都府京都市右京区花園土堂町10番地
(72)発明者　廣野　光明
京都府京都市右京区花園土堂町10番地
オムロン株式会社内
(72)発明者　乾　和志
京都府京都市右京区花園土堂町10番地
オムロン株式会社内
(72)発明者　小中　義治
京都府京都市右京区花園土堂町10番地
オムロン株式会社内
(74)代理人　100084548
弁理士　小森　久夫

審査官　久保　光宏

最終頁に続く

(54)【発明の名称】　プログラム制御装置および方法

1

(57)【特許請求の範囲】
【請求項１】　或るスレッドからの、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、コンテキストスイッチの発生有無を示すフラグをコンテキストスイッチが発生していない状態に設定する手段と、
前記フラグがコンテキストスイッチの発生していない状態に設定された後、スケジューラによってコンテキストスイッチが行われたとき、前記フラグをコンテキストスイッチが発生した状態に設定する手段と、
前記スレッドからの、コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、前記フラグの状態に応じた値を前記スレッドに返す手段と、
前記コンテキストスイッチが発生したとき、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの間の前記スレッドの処理を無効にする手段と、

2

前記或るスレッドの優先度を高い状態と低い状態とに交互に変更するとともに、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの処理時間を受け取って、前記スレッドの優先度が高い状態のとき、該スレッドの優先度が低い状態になるまでの残時間と前記処理時間とを比較し、前記残時間が前記処理時間より短いことを検知したときに、前記スレッドの優先度を低くす

る手段とを設けて成るプログラム制御装置。

【請求項２】　前記スレッドは、メモリのヒープ領域内の他のいずれかのオブジェクトから参照されているオブジェクトを検出し、当該オブジェクトを前記ヒープ領域内の所定領域に複写する複写方式のガベージコレクションスレッドである請求項１に記載のプログラム制御装置。

【請求項３】　前記スレッドは、メモリのヒープ領域内のどのオブジェクトからも参照されないオブジェクトのメモリ領域を他のオブジェクトのメモリ割り当て可能なフリー領域として解放することにより生じるフラグメントを解消するメモリコンパクションスレッドである請求項１に記載のプログラム制御装置。

【請求項４】　或るスレッドからの、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、コンテキストスイッチの発生有無を示すフラグをコンテキストスイッチが発生していない状態に設定するステップと、

前記フラグがコンテキストスイッチの発生していない状態に設定された後、スケジューラによってコンテキストスイッチが行われたとき、前記フラグをコンテキストスイッチが発生した状態に設定するステップと、

前記スレッドからの、コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、前記フラグの状態に応じた値を前記スレッドに返すステップと、

前記コンテキストスイッチが発生したとき、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの間の前記スレッドの処理を無効にするステップと、

前記或るスレッドの優先度を高い状態と低い状態とに交互に変更するとともに、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの処理時間を受け取って、前記スレッドの優先度が高い状態のとき、該スレッドの優先度が低い状態になるまでの残時間と前記処理時間とを比較し、前記残時間が前記処理時間より短いことを検知したとき、前記スレッドの優先度を低くするステップとから成るプログラム制御方法。

【請求項５】　或るスレッドからの、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、コンテキストスイッチの発生有無を示すフラグをコンテキストスイッチが発生していない状態に設定する処理プログラムと、

前記フラグがコンテキストスイッチの発生していない状態に設定された後、スケジューラによってコンテキストスイッチが行われたとき、前記フラグをコンテキストスイッチが発生した状態に設定する処理プログラムと、

前記スレッドからの、コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、前記フラグの状態に応じた値を前記スレッドに返す処理プログラムと、

前記コンテキストスイッチが発生したとき、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの間の前記スレッドの処理を無効にする処理プログラムと、

前記或るスレッドの優先度を高い状態と低い状態とに交互に変更するとともに、前記コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの処理時間を受け取って、前記スレッドの優先度が高い状態のとき、該スレッドの優先度が低い状態になるまでの残時間と前記処理時間とを比較し、前記残時間が前記処理時間より短いことを検知したとき、前記スレッドの優先度を低くする処理プログラムとを記録して成るプログラム記録媒体。

【発明の詳細な説明】

【０００１】

【発明の属する技術分野】この発明は、コンピュータのプログラム制御装置およびその方法に関する。

【０００２】

【従来の技術】先ず、本願発明の明細書において用いる各種用語の定義を以下に示す。尚、出典記号は次のとおりである。

【０００３】

ＪＩＳ：ＪＩＳ工業用語大辞典第４版（日本規格協会刊）

参１：情報技術用語事典（オーム社刊）

参２：情報処理用語大事典（オーム社刊）

参３：先端ソフトウエア用語事典（オーム社刊）

参Ａ：SUPER ASCII Glossary Help on Internet

(1) プロセス(process)：処理や過程を意味する一般用語（参２）

(2) スレッド(thread)：プロセスまたはタスクと呼ばれる１つの実行環境の中で並列実行可能な処理を複数に分割した場合に、プロセッサのスケジュール対象となる実行単位または制御フローのこと。（参３）

(3) コンテキスト(context)：（オブジェクト指向システムに関連して）メソッドを実行するために必要な情報を蓄えるオブジェクトをいう。コンテキストは、呼び出し先のコンテキスト、プログラムの入ったメソッドオブ

ジェクト、プログラムカウンタ、スタックポインタという情報、引数や一時変数を取るところ、および評価スタックから成立する。このような実行環境をオブジェクトとしてとるが、プロセスなどをサポートする高級言語の特徴でヒープ言語と呼ばれる。他方ＰＡＳＣＡＬやＡＬＧＯＬ６０では実行環境はスタックにとられＦＯＲＴＲＡＮでは固定領域にとられる。（参１）

(4) タスク(task)：多重プログラミング又は多重プロセッシングの環境において、計算機によって実行されるべき仕事の要素として制御プログラムによって取り扱われる命令の１つ以上の列（ＪＩＳ）

(5) ガベージ(garbage)：どこからも参照されていないが、生成されてしまったオブジェクト。ガベージはガーベジコレクタで回収する。（参１）

(6) ガベージコレクション(Garbage Collection)：（オブジェクト指向システムに関連して）メモリ管理の中心となるプログラムを言う。主記憶を使いきるとメインの計算を止め、ガーベジコレクタを動かして、もはや使われていないオブジェクトを全部集める方法である。この方法でガーベジコレクタが働くと、計算が止まってしまうため入出力が全く反応しなくなり、実時間の応答が必要な用途には使えない。（参１）

(7) インタプリタ(interpreter)：解釈実行を行う翻訳プログラム（ＪＩＳ）

(8) リアルタイム(real time)：計算機外部の他の処理との関係をもちながら、かつ外部の処理によって定められる時間要件に従って、計算機の行うデータの処理に関する用語。実時間。（ＪＩＳ）

(9) オブジェクト(object)：プロシジュア（定義した手続き）とデータの特性を結合させるエンティティ（実体）であり、これにより計算が行われ、局部的状態が蓄えられる。（参１）

(10) クラス(class)：同一の手続き群とデータ構造を持つオブジェクトの集合。（参３）

(11) ヒープ領域(heap area)：プログラム実行時に必要に応じて使用されるような作業領域。（参２）

(12) スケジュールする(scheduling)：ディスパッチされるべきジョブ又はタスクを選択すること。（ＪＩＳ）

(13) イベント(event)：ハードウエア／ソフトウエアが、自分自身の状態の変化を他のハードウエア／ソフトウエアに通知すること。一般にこの際の通知では、イベントの種類やハードウエア／ソフトウエアの状態を表す各種パラメータをメッセージとしてまとめ、相手に送信する。そしてイベントの通知を受けた側では、メッセージのパラメータなどから適切な処理を行う。（参Ａ）

(14) イベントフラグ(event flag)：タスクが１つまたは複数の事象の発生を待ち合わせるための機能とその事象を通知する機能とからなるタスク間同期通信機構。（参３）

(15) セマフォ(semaphore)：複数のプロセスやタスクを並列に処理するシステムで、各プロセス間，タスク間の同期やメッセージ制御、割込処理を行うための仕組み。（参３）

(16) 仮想マシーン（仮想機械）(virtual mathine)：複数の特定のプラットホーム（ＯＳやハードウエア）に組み込まれた、特定のプラットホームに依存しないアプリケーションプログラムを実行する環境。同じ仮想マシンさえ提供されていれば、プラットホームが変わっても同じアプリケーションプログラムが実行できる。

【０００４】(17) Java VM(Java Virtual Machine)：オペレーティングシステムに組み込まれた、Ｊａｖａアプリケーションプログラムを実行するための環境。一般的なプログラムは、ソースコードをコンパイルして、それぞれのオペレーティングシステムに最適化した実行コードを生成する、というスタイルを採る。Ｊａｖａで書かれたプログラムも同様の手順で作成するが、コンパイル後のコードは、特定のオペレーティングシステムに依存しない中間コードの形をとる。これをロードし、各オペレーティングシステムに合わせたコードに変換しながら実行するのがＪａｖａ仮想マシンの役目であり、同じ仮想マシンさえ提供されていれば、プラットホームが変わっても同じコードが実行できるようになっている。

【０００５】(18) フリー領域(free area)：ヒープ領域上の使用可能な領域、未使用領域。

【０００６】(19) ノーマルスレッド(normal thread)：リアルタイム性が要求されない処理を行うスレッド。

【０００７】(20) マークテーブル(mark table)：オブジェクトがどこからも参照されないか否かを調べるために存在するオブジェクトに１対１に対応する表。或るオブジェクトに参照があることが確かめられた場合、そのオブジェクトに対応する表の欄にマークを付ける。全ての参照関係を調べたときマークの無いオブジェクトは不要であるので取り除くことができる。

【０００８】(21) オブジェクトの寿命(life time of the object)：オブジェクトが生成され、消去されるまでの時間。

【０００９】(22) ライトバリア(write barrier)：オブジェクトへの参照関係の変更をチェックし、書き替えが起こった場合に何らかの処理をすること。本件の場合は、書き替えが起きたとき、書き替える参照が指しているオブジェクトに対応するマークテーブルの欄にマークを付ける。

【００１０】(23) スイープ(sweeping)：ヒープ領域上の不要なオブジェクトを取り除く処理。

【００１１】(24) オブジェクト生成(create the object)：新しいオブジェクトを生成すること。具体的には、ヒープ領域の一部をオブジェクトに割り当て、内容を初期化すること。

【００１２】(25) オブジェクト消去(delete the objec

t)：不要なオブジェクトを取り除くこと。具体的には、ヒープ領域に確保してある領域を解放すること。

【００１３】(26) 参照(reference)：或るオブジェクトＡが別の特定のオブジェクトＢにアクセスするためにオブジェクトＢを特定する情報。具体的には、オブジェクトＢを指すポインタまたはインデックス。

【００１４】(27) 参照変更(chenge the reference / reconnect the reference)：参照を現在のオブジェクトＢから別のオブジェクトＣに変更すること。

【００１５】さて、従来のシングルプロセッサのコンピュータシステムにおいて、オペレーティングシステム上で複数のスレッドを並行処理する場合、共有メモリを用いて排他制御するために、また複数のタスク間で同期をとるために、従来よりセマフォやイベントフラグなどを用いて排他制御が行われている。

【００１６】また、例えばプログラムを少量のメモリ環境下で動作させることなどを目的として、仮想記憶を行わないで単一のアドレス空間のメモリをプログラムの実行時に動的に割り当てる、いわゆる動的記憶管理が従来より行われている。このような動的記憶管理によれば、プログラムにより明示的にメモリ領域の解放を行わなければ、プログラムの実行過程において使用されなくなったメモリ領域が発生する。その結果、プログラムが使用できるフリー領域が次第に不足する。こうした問題を回避するために、使用されなくなったメモリ領域（ガベージ）を抽出し、これらを集めて（コレクトして）再び使用可能なフリー領域とする、ガベージコレクションと呼ばれる処理を自動的に行うようにしている。

【００１７】ここで従来のガベージコレクションの処理手順をフローチャートとして図４９に示す。ガベージコレクションのアルゴリズムにはマーク＆スイープ法、複写法、参照カウント法などの各種方法が考えられているが、ここではマーク＆スイープ法について例示する。図５４に示すように、まずガベージコレクションの途中でガベージコレクションスレッド以外の他のスレッドが実行されないように割り込みを禁止し、シングルスレッドモードにする(s201)。続いてメモリ上のガベージコレクションの対象となる領域（以下「ヒープ領域」という。）に割り当てられているオブジェクトにそれぞれ対応するマークの記憶領域（以下「マークテーブル」という。）をクリアする(s202)。続いてヒープ領域内に割り当てられているオブジェクトの参照関係を示す情報を基に何れかのオブジェクから参照されているオブジェクトを検出し、それらに対応するマークテーブル上の位置にマークを付ける処理を行う(s203)。この処理により、何れのオブジェクトからも参照されていないオブジェクトはもはや使われなくなったオブジェクトであり、それに相当するマークテーブルにはマークが付けられないことになる。従ってそのマークしていないオブジェクトを新たなオブジェクトの割当可能な領域、すなわちフリー領域として抽出する(s204)。例えばこのフリー領域をリスト構造のデータとして生成する。その後、割り込み禁止を解除し、マルチスレッドのモードに戻す(s205)。

【００１８】従来はこのようなガベージコレクションが、メモリのフリー領域が所定量まで減少した時点で自動的に起動されるようにしていた。

【００１９】また、ガベージコレクションを行ったとき、任意の大きさ（メモリサイズ）のオブジェクトが任意に解放されるのでフラグメントが発生する。そこで、連続したサイズの大きな領域がとれるように、オブジェクトの割当領域を例えば先頭から順次詰めるメモリコンパクション（以下単に「コンパクション」という。）を実行していた。

【００２０】

【発明が解決しようとする課題】上述した排他制御の機能をセマフォやイベントフラグなどのコンピュータ資源を用いることで実現している従来のシステムにおいては、その資源が他のスレッドなどで使用されている場合には、他のスレッドはその資源が解放されるのを待たなければならない。このような資源待ちの時間が生じると、リアルタイム性の要求されるシステムにおいては大きな障害となる。すなわち、資源待ち状態にあるスレッドは、その資源が解放されるまで、処理できずリアルタイムな応答が不可能となる。

【００２１】例えば上記従来のガベージコレクション（以下「ＧＣ」という。）の方法によれば、メモリ空間が広いほど、ガベージとしてコレクトしてもよい領域を見つけ出すのに時間がかかり、例えば６４～１２８ＭＢのヒープ領域で数秒間を要し、且つＧＣはフリー領域がある程度減少した時点で不定期に行われるので、リアルタイム性の要求されるシステムには用いることができなかった。

【００２２】リアルタイム性の要求されるシステムでは、あるタスクを実行中に何らかのイベント（割り込み）が発生すれば、そのイベントに応じた他のスレッドを処理することになるが、そのスレッドの切替に要する時間は、最悪値として例えば数十μｓｅｃ以下であることが要求される。ところが、上述したようにＧＣが何時起動されるか予測できず、一旦起動されれば、ＣＰＵは数秒間ＧＣに専念することになるため、その間リアルタイム処理は不可能となる。

【００２３】上述の問題はＧＣの場合に限らず、長時間の資源待ち状態が生じるシステムに共通の問題である。

【００２４】この発明の目的はセマフォやイベントフラグなどのコンピュータ資源をロックのメカニズムとして使用せずに、処理の排他性を保証することにより、上述の問題を解消することにある。

【００２５】ＧＣの他の方法として、従来の複写法をインクリメンタルに行うようにして、リアルタイム性を確保する方法が情報処理Vol.35 No.11 pp1008 ～pp1010に

示されている。この方法によれば、ＧＣの途中で中断を許すことができるので、時分割的に他のスレッドと並行処理することが一応はできる。しかし、この複写法をインクリメンタルに行うようにする方法では、複写中に他のスレッドがメモリを使用する訳にはいかないので、メモリを使用しない極一部のスレッドしか並行処理できない。また、複写法では、複写元と複写先のメモリ領域を確保しておかなければならないので、メモリの使用効率が低く、少量のメモリ環境下で動作させるシステムには向かない。

【００２６】また、マーク＆スイープ法でマルチスレッドに対応するために参照関係が変更される度にマークを付与する方法は、「On-the-fly GC 」という名前で、ＧＣ専用のＣＰＵに処理を割り当てるマルチＣＰＵ化のための方法が情報処理Vol.35 No.11 pp1006 ～pp1008に示されている。しかし、この方法では、オブジェクトが常に作成されて、且つ参照関係が変更されている場合に、既にマークしているツリーを何回も辿り直して新しいノードを見つけてマークしなければならず、マーク作業がいつまでも終わらない場合が生じたり、非常に長い時間がかかる、という問題があった。また、この方法ではスイープ中にシステムをロックする必要があった。

【００２７】そこで、この発明の他の目的はマーク＆スイープ法でありながらＧＣスレッドを他のスレッドと実質的に並行処理可能とし、ＧＣの任意の時点で中断しても短時間に確実にＧＣを完了できるようにしたインクリメンタルなＧＣを可能とすることにある。

【００２８】

【００２９】

【００３０】

【００３１】

【００３２】ここで、先行技術調査を行った際に発見した文献と本願発明との関係について示しておく。

【００３３】(1) Incremental Garbage Collection of Concurrent Objects for Real-TimeApplication

この論文はBaker が書いたという1978年のリアルタイムＧＣに対して、全体の処理時間から必要なＧＣの処理時間の割合を求めるものである。本願発明に係る課題を解決するものではない。

【００３４】(2) Distributed Garbage Collection for the Parallel Inference Machine:PIE64

ＧＣを行う領域を細かく分けることにより、個々の処理時間を短くしてリアルタイム性を向上する方法である。本願発明のように全ての領域を対象にするものとは基本的に異なる。また、スケジューリングについては述べられていない。

【００３５】(3) Garbage Collection in Distributed Enviroment

Baker の考えを発展させ、ネットワークの分散環境に適応したもの。ネットワーク固有の問題を解決しようとするものであり、本願発明とは異なる。この分散環境の考え方に、本願発明のスケジューラを組み合わせることにより、分散環境でより高度なリアルタイムＣＧを実現することも可能となる。

【００３６】(4) 特開平１－２２００３９号

システムコール発行時に、そのシステムコールにより起動されるタスクの優先度を制御するもの。優先度制御という点で共通点があるだけ。

【００３７】(5) 特開平３－２３１３３３号

オブジェクトのサイズを検出し、そのサイズに応じてワークエリアをメモリに確保することが一応示されている。

【００３８】(6) 電子情報通信学会誌Vol.80 No.6 pp586-592

マルチメディアオペレーティングシステムのコンセプトが示されている。

【００３９】(7) 日本ソフトウェア科学会第１２回Ｄ７－４

スタック上のオブジェクトの一番上にオブジェクトの大きさを書いておくことが示されている。すなわちオブジェクトのサイズを記憶しておく、という点でのみ関連がある。

【００４０】(8) 11TH Real-Time System Symposium "Incremental Garbage Collection ofConcurrent Object for Real-Time Applications"

オブジェクト間の参照ツリーに類似の考え方が示されている。

【００４１】(9) 情報処理学会第３８回全国大会5U-7

オブジェクトの寿命に関連してメモリ割当を行うことが示されている。

【００４２】(10)Lecture Notes in Computer Science 259 "Garbage Collection in a Distributed Environment"

アプリケーションプログラムインタフェースの呼出に応じてＧＣを制御することと、オブジェクト参照に関する技術思想が示されている。

【００４３】

【課題を解決するための手段】この発明の請求項１，４，５に係る発明は、或るスレッドからの、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、コンテキストスイッチの発生有無を示すフラグをコンテキストスイッチが発生していない状態に設定し、前記フラグがコンテキストスイッチの発生していない状態に設定された後、スケジューラによってコンテキストスイッチが行われたとき、前記フラグをコンテキストスイッチが発生した状態に設定するようにし、前記スレッドからの、コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しに応答して、前記フラグの状態に応じた値を前記スレ

ッドに返す。

【００４４】これにより、コンピュータ資源をロックのメカニズムとして使用せずに、処理の排他性を保証する。すなわち或るスレッドＡからのコンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェース（以下「ＡＰＩ」という。）の呼び出しからコンテキストスイッチ発生有無検出の終了を依頼するＡＰＩの呼び出しまでの間で行われたスレッドＡの処理の途中で、コンテキストスイッチがあったか否かを、そのスレッドＡで分かるようにする。もしコンテキストスイッチが発生しなければ、スレッドの切替は行われておらず、排他性が保たれている（例えば上記ＡＰＩを発行したスレッドＡが使用しているメモリ内容が他のスレッドＢにより書き替えられていない）ことが分かる。逆に、コンテキストスイッチが発生していれば、その間のスレッドの処理を無効にして、例えばその処理を再度実行するなどの方法によって高い応答性を保ちながら、排他制御を行うことができるようにする。

【００４５】また、或るスレッドの優先度を高い状態と低い状態とに交互に変更するようにしておき、コンテキストスイッチ発生有無検出の開始を依頼するアプリケーションプログラムインタフェースの呼び出しから前記コンテキストスイッチ発生有無検出の終了を依頼するアプリケーションプログラムインタフェースの呼び出しまでの予定されている処理時間を受け取って、前記スレッドの優先度が高い状態のとき、該スレッドの優先度が低い状態になるまでの残時間と前記処理時間とを比較し、前記残時間が前記処理時間より短いことを検知したときに、前記スレッドの優先度を低くする。

【００４６】これにより、上記処理時間内に終了できない状態は、その処理が行われることがないため、その分のＣＰＵパワーを無駄にすることがなく、全体の処理を進められる。

【００４７】請求項２に係る発明は、メモリのヒープ領域内の他のいずれかのオブジェクトから参照されているオブジェクトを検出し、当該オブジェクトを前記ヒープ領域内の所定領域に複写する複写方式のガベージコレクションスレッドの際に、上記のコンテキストスイッチの検出による処理を行う。これにより、システムをロックすることなく、ＧＣを開始することができ、リアルタイム性を保証することができる。

【００４８】請求項３に係る発明は、メモリのヒープ領域内のどのオブジェクトからも参照されないオブジェクトのメモリ領域を他のオブジェクトのメモリ割り当て可能なフリー領域として解放することにより生じるフラグメントを解消するメモリコンパクションを行う際に、上記のコンテキストスイッチの検出による処理を行う。これにより、システムをロックすることなく、メモリコンパクションを開始することができ、リアルタイム性を保証することができる。

【００４９】
【００５０】
【００５１】
【００５２】
【００５３】
【００５４】
【００５５】
【００５６】
【００５７】
【００５８】
【００５９】
【００６０】
【００６１】
【００６２】
【００６３】
【００６４】
【００６５】
【００６６】
【００６７】
【００６８】
【００６９】
【００７０】
【００７１】
【００７２】
【００７３】
【００７４】
【００７５】

【発明の実施の形態】この発明の実施形態であるプログラム制御装置およびメモリ割り当て装置の構成を図１～図５３を参照して順次説明する。

〔第１の実施形態〕

【００７６】図１は装置のハードウエアの構成を示すブロック図である。装置は基本的にＣＰＵ１とオブジェクト群を生成するヒープ領域およびマークテーブル等を記憶するメモリ２、および外部との入出力を行うＩ／Ｏ３とから構成する。また、外部から必要なプログラムをメモリにロードする場合は、ＣＤ－ＲＯＭ読取インタフェース４を用い、プログラムが予め書き込まれたＣＤ－ＲＯＭ５を読み取るようにする。このＣＤ－ＲＯＭが本願発明に係るプログラム記録媒体に相当する。

【００７７】図２はソフトウエアの構成を示すブロック図である。同図において、カーネル部分はＣＰＵやメモリを資源として管理し、時分割によるマルチスレッドの機能を実現する。ＶＭ（仮想マシーン）部分はプログラムとカーネルとのインタフェースを行うソフトウエアであり、ここではアプリケーションプログラムから見てＶＭ以下の階層全体が例えばＪａｖａ仮想マシンとして作用させる。（ＪａｖａはSun Microsystems社の商標）この場合、カーネルとＶＭ部分がＪａｖａＯＳを構成する。このＶＭにはプログラムがバイトコード等の中間コ

ードで与えられるとき、これを解釈するインタプリタとその解釈に応じて呼び出されるプログラムモジュール等を含む。同図におけるプログラムは中間コードによる各種スレッドであり、上記インタプリタを介して、内部のプログラムモジュールを実行する。

【００７８】図３はメモリのヒープ領域に生成されるオブジェクトの参照関係およびスタックとの関係を示す図である。ヒープ領域に対してオブジェクトを生成した際、或るオブジェクトから他のオブジェクトへの参照関係は同図に示すようにルートノードから延びるツリー構造を成す。例えばグローバル変数を宣言すれば、その変数に対応するオブジェクトが生成される。またスレッド毎に引き数領域、戻り番地、ローカル変数、作業領域等の情報を記憶するスタックが生成され、例えば図中矢印で示すようなスタック上のローカル変数からツリー上のグローバル変数への参照関係なども記憶される。これらのスタックはヒープ領域外の所定領域に格納される。

【００７９】図４は図２に示したソフトウエアの構成をブロック図として詳細に示したものである。同図において、ＧＣモジュール１０はＧＣを行うための各種処理のプログラムモジュールであり、ＧＣスレッド１１はこれらのモジュールを呼び出すことによって、ＧＣを実行させる。また、インタプリタ１２は、この例で、「スレッド４」からオブジェクト生成依頼があったとき、ＧＣモジュール１０の「オブジェクト生成」を呼び出し、またオブジェクト間の参照関係の変更依頼があったとき、ＧＣモジュール１０の「マーク付与」を呼び出す。

【００８０】なお、図４に示した例では、各スレッドが中間コード（例えばＪａｖａアプレットの場合バイトコード）で表されているが、中間コードをＶＭに対するネイティブコードに変換するコンバイラを設けてもよい。（例えばＪａｖａの場合、ＪＩＴ(Just-In-Time コンバイラ）等を設ける。）この場合、ネイティブコードで記述したスレッドであるので、図４に示したインタプリタ１２を介さずにＧＣモジュール１０を直接アクセスすることになる。

【００８１】図５はＧＣを行うことによって、ヒープ領域内に生じるフラグメントを解消するためのコンパクションを行った例を示す図である。図中ハッチング部分がオブジェクトであり、これをコンパクションすることによって、（Ｂ）に示すようにフラグメントが解消されて、連続したメモリ空間が広がることになる。

【００８２】上記コンパクションは図４に示したＧＣモジュールの「コンパクション」のプログラムにより行う。

【００８３】図６はコンテキストスイッチ発生有無を検出するためのＡＰＩの使用例を示す図である。同図の（Ａ）に示すように、処理１を実行する前にコンテキストスイッチ発生有無の検出開始を依頼するＡＰＩ＃Ａを発行してから処理１を実行する。処理１の終了後、コンテキストスイッチ発生有無検出の終了を依頼するＡＰＩ＃Ｂを発行する。（Ａ）に示す例ではこの間にコンテキストスイッチが発生していないので、そのまま次の処理２を行う。もし、（Ｂ）に示すように、処理１の途中でスレッド＃２の処理が行われれば、すなわちコンテキストスイッチが発生していれば、ＡＰＩ＃Ｂの発行後、処理１を破棄する。たとえばメモリの領域Ａの内容を領域Ｂにコピーするような処理で、コピー処理中にコンテキストスイッチが発生した場合、領域Ａの内容が変わって領域ＡとＢの内容が不一致となる場合がある。不一致ではコピーしたことにならないので、領域Ｂを無効にする。このことは、最初から処理が行われなかったのと同じであり、処理自体を破棄したことに他ならない。

【００８４】図７は上記の処理をフローチャートとして示したものである。まず、ＡＰＩ＃Ａを発行してから処理１を実行する（ｓ１→ｓ２）。この処理１が終了した後、ＡＰＩ＃Ｂを発行して、その戻り値を取得する（ｓ３）。戻り値がコンテキストスイッチの発生を表す場合、処理１を破棄して、再び処理１を実行する（ｓ４→ｓ５→ｓ１→ｓ２）。戻り値がコンテキストスイッチの非発生を表す場合、処理を終了する。このように所定の期間内でのコンテキストスイッチの発生有無が分かるので、コンテキストスイッチが発生すれば、その間の処理を破棄し無効とすることによって、システムを現実にはロックしていないにも拘らず排他的制御を行うことが可能となる。

【００８５】図８は上記ＡＰＩ＃ＡおよびＡＰＩ＃Ｂのカーネルにおける処理手順を示すフローチャートである。ＡＰＩ＃Ａの発行（システムコール）があれば、コンテキストスイッチの発生有無を示すフラグをクリアする。また、ＡＰＩ＃Ｂが発行されると、上記フラグの状態を戻り値としてスレッドに返す。

【００８６】図９はコンテキストスイッチの処理手順を示すフローチャートである。コンテキストスイッチがスケジューラによって行われると、上記フラグをセットしてからコンテキストスイッチを実行する。すなわちスイッチ前のスレッドの実行状態をコンテキストとして格納するとともに、スイッチ後のスレッドのコンテキストを読み出してＣＰＵのレジスタ等に設定する。

【００８７】図１０は上記コンパクションの処理手順を示すフローチャートである。先ず、ヒープ領域内の先頭のオブジェクトを指定し（ｓ１１）、そのオブジェクトをヒープ領域の先頭にコピーするための領域を確保し（ｓ１２）、コピー中に他のスレッドによってその領域に何らかのデータが書き込まれないようにしてから上記ＡＰＩ＃Ａを発行する（ｓ１３）。そして、図５に示したように、ヒープ領域内のオブジェクトを先頭から順次空き領域へコピーすることによって詰めていく（ｓ１４）。１つのオブジェクトについてのコピーが終われば、上記ＡＰＩ＃Ｂを発行する（ｓ１５）。このことに

より、ＡＰＩ＃Ａを発行してから、ＡＰＩ＃Ｂを発行するまでの期間にコンテキストスイッチが発生したか否かがＡＰＩ＃Ｂの戻り値として得られる。もしコンテキストスイッチが発生していれば、今回コピーを行ったオブジェクトを既に確保している領域に再びコピーする（ｓ１６→ｓ１３→・・・）。もしコンテキストスイッチが発生していなければ、次のオブジェクトについて同様に処理を行う（ｓ１６→ｓ１７→ｓ１８→・・・）。このようにシステムをロックすることなく、しかも他のスレッドとともに、コンパクションを同時に行うことが可能となる。

【００８８】上述の例ではマーク＆スイープ法によるＧＣにおけるコンパクションに適用したが、複写法によるＧＣに適用する場合、図１１および図１２に示す処理を行う。

【００８９】図１１は上記複写法によるＧＣのフローチャートである。先ず、オブジェクトの参照関係を表すツリー構造のデータのルートノードへポインタを移動し（ｓ２１）、そのルートノードに相当する、Ｆｒｏｍ領域にあるオブジェクトをＴｏ領域に複写する（ｓ２２）。（ヒープ領域はＦｒｏｍ領域とＴｏ領域に分けられ、Ｆｒｏｍ領域内の残すべきオブジェクトのみをＴｏ領域に複写することによってＴｏ領域にガベージのないオブジェクトだけを再構成する。次回は現在のＦｒｏｍ領域をＴｏ領域とし、Ｔｏ領域をＦｒｏｍ領域として、その操作を交互に繰り返すのが、複写法によるガベージコレクションである。）その後、ツリーを辿って、参照関係にある次のオブジェクトへポインタを移動させ（ｓ２３）、そのオブジェクトをＴｏ領域に複写する（ｓ２４）。この処理をツリーで辿れるすべてのオブジェクトについて行う。

【００９０】図１２は図１１における複写処理の内容を示すフローチャートである。先ず、複写すべきオブジェクトをＴｏ領域内の所定位置にコピーするための領域を確保し、コピー中に他のスレッドによってその領域に何らかのデータが書き込まれないようにし、上記ＡＰＩ＃Ａを発行してから（ｓ３１）、オブジェクトをＦｒｏｍ領域からＴｏ領域に複写する（ｓ３２）。その後、上記ＡＰＩ＃Ｂを発行する（ｓ３３）。このことにより、ＡＰＩ＃Ａを発行してから、ＡＰＩ＃Ｂを発行するまでの期間にコンテキストスイッチが発生したか否かがＡＰＩ＃Ｂの戻り値として得られる。もしコンテキストスイッチが発生していれば、今回複写を行ったオブジェクトを既に確保している領域に再び複写する（ｓ３４→ｓ３１→・・・）。

【００９１】図１３および図１４はコンテキストスイッチの発生有無を検出して排他性を確保するのではなく、複写しようとするオブジェクトが複写前に書き替えられたか否かを検出して排他性を確保する場合の処理手順を示すフローチャートである。

【００９２】複写を行う場合、先ず複写すべきオブジェクトをＴｏ領域内の所定位置にコピーするための領域を確保し、コピー中に他のスレッドによってその領域に何らかのデータが書き込まれないようにしてから、図１３に示すようにＡＰＩ＃Ｃを発行する（ｓ４１）。このＡＰＩは指定したメモリ領域ない対する書き込みが発生したか否かの検出を依頼するアプリケーションプログラムインタフェースである。続いてオブジェクトをＦｒｏｍ領域からＴｏ領域に複写する（ｓ４２）。その後、上記ＡＰＩ＃Ｄを発行する（ｓ４３）。このＡＰＩは上記ＡＰＩ＃Ｃが呼び出されてから、このＡＰＩ＃Ｄが呼び出されるまでの間に指定メモリ領域に何らかのデータの書き込みが発生したか否かが戻り値として返されるアプリケーションプログラムインタフェースである。したがって、複写しようとするオブジェクトのメモリ領域を指定してＡＰＩ＃Ｃを発行し、ＡＰＩ＃Ｄも戻り値を見れば、そのオブジェクトが参照されたか否かが判る。オブジェクトが参照されていれば、内容が書き変わっている可能性があるので、そのオブジェクトの複写をやり直す（ｓ４４→ｓ４１→・・・）。

【００９３】図１４は上記ＡＰＩ＃ＣおよびＡＰＩ＃Ｄのカーネルにおける処理手順を示すフローチャートである。ＡＰＩ＃Ｃの発行（システムコール）があれば、所定にワークエリアをクリアし、ＡＰＩ＃Ｃのパラメータで指定された領域がライトされたときに例外が発生するようにＭＭＵ（Memory Management Unit）を設定する。また、ＡＰＩ＃Ｄが発行されると、上記パラメータで指定された領域がライトされたときに例外が発生しないように、ＭＭＵに対する上記設定を解除する。そして上記ワーク領域にセットされるフラグの状態を戻り値としてスレッドに返す。

【００９４】図１５は上記例外発生時のＭＭＵの処理内容を示すフローチャートである。例外が発生すると、上記ワーク領域内の参照フラグをセットする。

【００９５】なお、上述の例では複写法によるＧＣの場合について示したが、マーク＆スイープ法におけるコンパクションの場合にも同様に適用できる。

【００９６】〔第２の実施形態〕図１６および図１７はインクリメンタルにガベージコレクションできるＧＣスレッドの優先度を自動的に切り替えるようにした場合の例を示す図である。

【００９７】図１６の（Ａ）に示すように、ＧＣスレッドの優先度を、高優先度時間は高め（例えば最高優先度にし）、低優先度時間は低く（例えば最低優先度）する動作を交互に行う。

【００９８】同図の（Ｂ）はＧＣスレッド以外の中程度の優先度のスレッドを同時に行った場合の例について示している。すなわちＧＣスレッドの優先度が低いとき、それより優先度の高いスレッドがＲｅａｄｙ状態となれば、コンテキストがスイッチされ、そのスレッドの実行

中にＧＣスレッドの優先度が高くなれば、そのＧＣスレッドにコンテキストがスイッチされ、そのＧＣスレッドの処理が中断すると、上記のＧＣスレッド以外のスレッドの処理を行うことになる。このようにすれば、一定周期で必ずＧＣスレッドが実行されるため、フリー領域が慢性的に不足状態となることが防止され、常に高いパフォーマンスを維持することができる。

【００９９】図１７はスレッドの優先度の切替に関するカーネルの行う処理手順を示すフローチャートである。優先度の値は複数段階あり、その値の設定は対応するＡＰＩの発行により行えるようにしている。ここでは、ＧＣスレッドの２つの優先度を設定する際、高優先度の値を設定するＡＰＩが発行されると、図１７の（Ａ）に示すように、その値をＧＣスレッドの高優先度の値として設定する。同様に、低優先度の値を設定するＡＰＩが発行されると、（Ｂ）に示すように、その値をＧＣスレッドの低優先度の値として設定する。また、ＧＣスレッドの高優先度の時間を設定するＡＰＩが発行されると、同図の（Ｃ）に示すように、その値を設定し、同様に低優先度の時間を設定するＡＰＩが発行されると、同図の（Ｄ）に示すように、その値を設定する。

【０１００】図１７の（Ｅ）は、カーネルのスケジューラに対する処理手順を示すフローチャートである。ここでは、初期状態で、高優先度のキューにＧＣスレッドが資源の割当を受けようとする待ち行列に入っているものとし、先ず、そのデータ（ＧＣスレッドを識別するデータ）を高優先度のキューから取り出して低優先度のキューに挿入する（ｓ５１）。そしてスケジューラは、既に設定されている低優先度時間だけ時間待ちを行い（ｓ５２）、続いて低優先度のキューからＧＣスレッドを識別するデータを取り出して高優先度のキューに挿入する（ｓ５３）。そしてスケジューラは、既に設定されている高優先度時間だけ時間待ちを行う（ｓ５４）。以上の処理を繰り返すことによって、スケジューラの動作により図１６（Ａ）に示したようにＧＣスレッドの優先度が交互に切り替わる。ＧＣスレッド以外のスレッドについては、そのスレッドの優先度に対応するキューの待ち行列に応じて従来と同様のスケジューリングが行われ、図１６の（Ｂ）に示したように、コンテキストスイッチがなされる。

【０１０１】図１８は上記のＧＣスレッドの高優先度時間をＡＰＩによって設定可能とした場合について示している。また、図１９はそのＡＰＩの呼び出しに応じたカーネルにおける処理手順を示すフローチャートである。このＡＰＩの発行（システムコール）があれば、上記の高優先度時間をそのＡＰＩのパラメータによって設定（登録）する。

【０１０２】図２０は上記のＧＣスレッドの高優先度時間を設定可能とするＡＰＩを使うプログラムの例を示すフローチャートである。先ず、ループの１回目の示すフラグの状態を見る（ｓ６１）。最初はＯＦＦ状態であるので、このフラグをＯＮし（ｓ６２）、現在のフリー領域を調べ、それを記録する（ｓ６３）。初めはＧＣの高優先度時間と低優先度時間は予め定められたデフォルト値である。スケジューラにより、一定時間停止した後（ｓ６４）、今度は、前回に調べたフリー領域の容量と現在のフリー領域の容量との大小比較を行い（ｓ６５）、フリー領域の容量が増加していれば、ＧＣスレッドの高優先度時間を短くし（ｓ６６→ｓ６７）、フリー領域の容量が減少していれば、ＧＣスレッドの高優先度時間を長くする（ｓ６８）。以上の処理を繰り返す。このことによって、ＧＣの優先度が動的に自動調整される。

【０１０３】図２１はＧＣスレッドの高優先度時間と低優先度時間とによる周期を設定可能とした場合について示している。（Ａ）は周期が長い場合、（Ｂ）は短い場合である。

【０１０４】図２２は上記周期を設定可能とする周期設定ＡＰＩの呼び出しに応じたカーネルにおける処理手順を示すフローチャートである。この周期設定ＡＰＩの発行があれば、現在設定されている高優先度時間および低優先度時間の値を読み取り、その割合（比率）を計算する（ｓ７１→ｓ７２→ｓ７３）。その後、この周期設定ＡＰＩのパラメータで指定された周期の値に応じて高・低優先度時間を再計算する（ｓ７４）。そして、その高優先度時間および低優先度時間を設定更新する（ｓ７５→ｓ７６）。

【０１０５】たとえば連続的なパルスを処理するような、連続してＣＰＵ資源が必要なアプリケーションプログラムの場合には、ＧＣスレッドの周期が短くなるように周期設定ＡＰＩを発行し、通常のアプリケーションプログラムのように断続的にＣＰＵ資源を利用するアプリケーションプログラムでは周期が長くなるように周期設定ＡＰＩを発行する。

【０１０６】〔第３の実施形態〕図２３および図２４は必要な時点でＧＣスレッドを実行し、且つリアルタイム性を確保するようにした例であり、図２３に示す例では、スレッド１とスレッド３がリアルタイム性の要求されるスレッド、スレッド２がリアルタイム性の要求されないスレッド（ノーマルスレッド）である。また、スレッド３がＧＣスレッドである。メモリのフリー領域が多い通常状態で、スレッド２が実行されているときにイベント１が発生すれば処理がスレッド１に移され、イベント１の処理に起因するスレッド１の処理が終了すればスレッド２へ戻る。同様に、イベント２が発生すればスレッド３に処理が移る。もしスレッド２の処理によってフリー領域が予め定めた警告レベルまで低下したとき、スレッド２の処理を中断して、スレッド３のＧＣスレッドを実行する。この処理によってフリー領域が確保されるとスレッド２の処理へ戻る。もしＧＣスレッドの処理途

中でイベント１が発生すれば、ＧＣの途中であってもスレッド１へ処理が移る。このようにリアルタイム性の要求されるスレッドでは、一般に生成されるオブジェクトの量が少なく、大体の予測が可能であるので、それに応じて上記警告レベルを設定しておけば、スレッド２に処理によって、フリー領域が大幅に減少してスレッド１やスレッド３のリアルタイム性の要求される処理が実行できなくなるといった、不都合が回避できる。

【０１０７】図２４は上記コンテキストスイッチの処理を行うスケジューラの処理手順を示すフローチャートである。先ずイベントの発生有無を検知し（ｓ８１）、イベントが発生していれば、対応するリアルタイムスレッドにコンテキストをスイッチする（ｓ８２）。イベントが発生していなくて、フリー領域が警告レベルより多ければ、ノーマルスレッドのうち優先度の最も高いもの（図２３に示した例ではスレッド２）にコンテキストをスイッチする（ｓ８３→ｓ８４）。もしフリー領域が警告レベルより少なくなれば、ＧＣスレッドにコンテキストをスイッチする（ｓ８５）。

【０１０８】〔第４の実施形態〕図２５はコンテキストスイッチの検出を依頼するＡＰＩにおける強制コンテキストスイッチの例を示している。先に示した例では、ＡＰＩ＃Ａを発行してから、ＡＰＩ＃Ｂを発行するまでの間にコンテキストスイッチが発生したか否かが判るようにしたものであったが、この図２５に示す例は、ＧＣの優先度を高・低交互に切り替える場合に、上記の排他制御のためのＡＰＩを発行している間にコンテキストスイッチが発生することを予測して、無駄な処理を行わないようにしたものである。すなわち、高優先度ＧＣスレッドを実行中にＡＰＩ＃Ａ′を発行した際、ＡＰＩ＃Ａ′は高優先度時間が終了するまでに、必要な処理が完了するか否かを判定して、もし完了しなければ、その時点でＧＣスレッドの優先度を低くする。図２５に示した例では、ＧＣスレッドの優先度が低くなったことにより、ＧＣスレッド以外の中優先度のスレッドにコンテキストスイッチされることになる。このことにより無駄な処理が避けられる。

【０１０９】図２６は上記ＡＰＩ＃Ａ′の呼び出しに応じたカーネルにおける処理手順を示すフローチャートである。このＡＰＩの呼び出しがあれば、コンテキストスイッチフラグをクリアし（ｓ９１）、ＧＣスレッドの高優先度時間の残時間を取得する（ｓ９２）。そして、このＡＰＩ＃Ａ′のパラメータである排他時間（ＡＰＩ＃Ａ′を発行してから、ＡＰＩ＃Ｂを発行するまでの時間）と上記残時間との大小比較を行う（ｓ９３）。もし、残時間が排他時間より短ければＧＣスレッドの優先度を強制的に低くする（ｓ９４）。なお、ＡＰＩ＃Ｂ呼び出しによる処理内容とコンテキストスイッチの処理内容は図８および図９に示したものと同様である。

【０１１０】〔第５の実施形態〕図２７はメモリ（ヒープ領域）使用量に応じてＧＣのアルゴリズムを切り替えるための処理手順を示すフローチャートである。先ず、メモリ使用量の値とそれに応じてどのアルゴリズムでＧＣを行うかを示すしきい値を設定し（ｓ１０１）、メモリ使用量を取得する（ｓ１０２）。これはヒープ領域内に生成されたオブジェクトのメモリサイズの合計値である。メモリ使用量がしきい値ｔｈ１を超えた場合、ＧＣアルゴリズム（１）の手順でＧＣを行う（ｓ１０３→ｓ１０４）。メモリ使用量がしきい値ｔｈ２を超えたなら、ＧＣアルゴリズム（２）の手順でＧＣを行う（ｓ１０５→ｓ１０６）。以下同様である。ここで、しきい値ｔｈ１はしきい値ｔｈ２より小さく、ＧＣアルゴリズム（１）としては図１６に示した、ＧＣスレッドの優先度の高低を交互に繰り返す処理を行う。しきい値の高いときに行うＧＣアルゴリズムとしては、図２３に示した不定期のＧＣを実行する。ここでＧＣ自体は例えばマーク＆スイープ法により行う。通常、前者の場合にはＣＰＵ負荷が軽く、専らこの方法によりＧＣが行われるが、ノーマルスレッドによって短時間に大量のメモリが使用された場合には、図２３に示した方法によりＧＣを重点的に行う。このことによって常に広いフリー領域を確保し、且つリアルタイム性を維持する。

【０１１１】〔第６の実施形態〕図２８～図３０はオブジェクト生成時のヒープ領域に対する新たなオブジェクトの割り当てサイズを定めるための処理に関する図である。一般にプログラムの実行により生成されるオブジェクトのサイズの分布は図２８に示すように略正規分布を示す。その中心値は３２と６４バイトの間に納まる程度である。そこで、図２９の（Ａ）に示すように、この中心値より大きなサイズで、且つ２の巾乗バイトの整数倍のサイズをオブジェクトの割り当てサイズとする。従来は同図の（Ｂ）に示すように、生成すべきオブジェクトのサイズの量だけ任意に割り当てられていたため、そのオブジェクトの消去の際、不揃いのサイズのフラグメントが生じる。本願発明によれば、新たに生成されるオブジェクトのサイズが上記固定値の整数倍であるため、メモリ領域の再利用性が高まり、メモリ使用効率が全体に高まる。しかも場合によってはコンパクションが不要となる。

【０１１２】図３０はそのオブジェクト生成時の処理手順を示すフローチャートである。先ず、これまでに生成したオブジェクトのサイズの発生頻度の分布データを求める（ｓ１１１）。既に前回までに分布データを求めている場合には、それを更新する。続いて、今回割り当てるべきメモリの空いている領域で、且つ生成しようとするオブジェクトのサイズより大きな領域を探し、上記の固定サイズの整数倍のメモリ領域にオブジェクトを割り当てる（ｓ１１２→ｓ１１３→ｓ１１４→ｓ１１５）。上記２の巾乗バイトはシステム定数であるが、必ずしもこの値を固定サイズとする必要はなく、任意である。固

定サイズをもし大き過ぎる値に採れば、サイズの大きな領域に小さなオブジェクトが割り当てられる場合が増えることになり、逆に、固定サイズを小さ過ぎる値に採れば、オブジェクトの生成に再利用できない領域が増えることになる。分布データを基に上記固定サイズを決定する場合には、メモリの使用効率が最適となるように決定すればよい。また、最適値でなくても、例えば２の巾乗の値を採れば、アドレス決定がやり易くなるという効果を奏する。

【０１１３】図３１は上記オブジェクトのサイズの発生頻度分布データを求めるプログラムモジュール、およびオブジェクトの割り当てサイズを決定するプログラムモジュールの処理内容を示すフローチャートである。先ず、オブジェクトのサイズの発生頻度分布データを求めるプログラムモジュールをロードし（ｓ１２１）、そのプログラムモジュールを起動する。すなわち一定時間が経過するまで（たとえばアプリケーションプログラムの起動・終了が１０回行われるまで、またたとえば２４時間経過するまで）、生成された各オブジェクトのサイズをサイズ毎に計数し、その分布データを求める（ｓ１２２→ｓ１２３）。その後、その分布データをシステムに登録し（ｓ１２４）、オブジェクトのサイズの発生頻度分布データを求めるプログラムモジュールをアンロードする（ｓ１２５）。続いて固定サイズを決定するプログラムモジュールをロードし（ｓ１２６）、そのプログラムモジュールを実行する。すなわちシステムの登録された分布データを取得し、その分布中心値より大きなサイズで、且つたとえば２の巾乗バイトを固定サイズとして決定し、その値をシステムに登録する（ｓ１２７→ｓ１２８）。その後、固定サイズを決定するプログラムモジュールをアンロードする（ｓ１２９）。

【０１１４】このように、一度オブジェクトのサイズの分布が検知された後、および固定サイズが決定された後は、それらのためのプログラムモジュールをアンロードすることによって、メモリ領域とＣＰＵパワーの無駄を無くす。

【０１１５】〔第７の実施形態〕図４９は上記固定サイズをＡＰＩによって設定する例を示している。図に示すように、固定サイズ設定処理では、固定サイズを引数として固定サイズ設定ＡＰＩを呼び出す。呼び出されたＡＰＩでは、引数を固定サイズとしてシステムに登録する。オブジェクト生成時にはオブジェクトのサイズと固定サイズを比較し（ｓ２１１）、固定サイズ以下であれば、ヒープ上の固定サイズの空き領域を探し、見つかった領域をオブジェクトに割り当てる（ｓ２１２→ｓ２１３）。固定サイズを超える場合は、ヒープ上のオブジェクトサイズより大きな空き領域を探し、見つかった領域をオブジェクトに割り当てる（ｓ２１４→ｓ２１５）。

【０１１６】〔第８の実施形態〕図５０は上記固定サイズをＡＰＩによって設定する他の例を示している。図５０に示すように、この例では、まずオブジェクトサイズの分布データをオブジェクトサイズ分布設定ＡＰＩを呼び出して設定する。この分布データは予め所定時間所定のアプリケーションを実行して測定したものである。オブジェクトサイズ分布設定ＡＰＩは呼び出しに応答して、引数をオブジェクトサイズ配列変数にセットする。続いて、その中心値より大きなサイズで、且つ２の巾乗バイトを固定サイズとして決定し、これを固定サイズ変数にセットする。

【０１１７】〔第９の実施形態〕図５１～図５３はオブジェクトの割り当てサイズを予めいくつかのサイズに定めておく例を示す図である。図５１は事前に所定のアプリケーションを所定時間実行して測定したオブジェクトサイズの分布および分割領域の集合を表している。予めヒープ領域を分割する場合、実際のオブジェクトサイズの分布を測定し、その分布に近くなるように、分割サイズと数を定める。たとえば、ヒープ領域が２ＭＢの場合、分割サイズ指定変数を

集合No.n	バイトk	個数m
1	64	5000
2	256	10000
3	1k	10000
4	4k	5000
5	32k	500

とする。

【０１１８】上記分割サイズを設定する場合、図５２に示すように、分割サイズ設定ＡＰＩを呼び出して行う。呼び出された分割サイズ設定ＡＰＩは図５３に示すように、引数を分割サイズ指定変数にセットし、それに応じてヒープ領域を分割する。すなわち、まずループカウンタｎを０とし（ｓ２２１）、ｎ番目の分割サイズ指定変数からサイズｋと個数ｍを取得する（ｓ２２２）。続いてヒープ領域からサイズｋの領域をｍ個に分割し（ｍ個分確保し）、リストに登録する（ｓ２２３）。この処理をループカウンタｎを１インクリメントしながら繰り返し、全てのサイズの分割を行う（ｓ２２５→ｓ２２２→・・・）。なお、上記の例で、サイズが３２ｋバイトを超えるオブジェクトを生成する場合は、ヒープ領域内の上記分割された領域以外の領域に割り当てる。

【０１１９】〔第１０の実施形態〕図３２～図３４はオブジェクトの寿命を考慮してＧＣの効率を高めるための処理を行う例を示す図である。図３２の（Ａ）に示す例では、ヒープ領域として、短寿命のオブジェクトを生成する領域と長寿命のオブジェクトを生成する領域とに分け、クラスは長寿命領域に確保する。尚、クラスはその他の固定領域に確保してもよい。そして、クラスにはオブジェクトを生成するためのテンプレートとしての定義情報以外に、生成するオブジェクトの寿命を示す寿命フラグを持たせる。この寿命フラグはクラスの生成時に自

動的に生成する。同図の（Ｂ）は従来のヒープ領域に対するオブジェクトの割り当て例を示す図である。

【０１２０】図３３はオブジェクトの消去の手順を示すフローチャートである。同図に示すようにオブジェクトを消去した際に、そのオブジェクトの寿命が長いか短いかを判定し、寿命が短ければ、そのオブジェクトのクラスの寿命フラグを短寿命にセットする。例えばオブジェクトの領域内にそのオブジェクトが生成された時刻を格納しておき、そのオブジェクトを消去する時刻との差によって、そのオブジェクトの寿命を求める。上記時刻は例えばＧＣの回数を単位としてもよい。

【０１２１】図３４はオブジェクトの生成の処理手順を示すフローチャートである。クラスの寿命フラグが短寿命を示していれば、短寿命領域にオブジェクトを生成し、そうでなければ、長寿命領域にオブジェクトを生成する。

【０１２２】このようにオブジェクトの寿命に応じてメモリの割り当て領域を区分することによって、長寿命領域ではフラグメントを大幅に低下でき、メモリの使用効率が向上する。また、例えば寿命領域についてＧＣを重点的に行うことにより、ＧＣに消費されるＣＰＵパワーを小さくすることができる。

【０１２３】〔第１１の実施形態〕次に、マーク＆スイープ法によるインクリメンタルＧＣについて図３５～４８を参照して説明する。

【０１２４】図３５はマーク＆スイープ法によるＧＣの全体の処理手順を示すフローチャートである。このＧＣはマークテーブルのクリア、上記ツリー探索によるマーク付与およびオブジェクトの消去（スイープ）を繰り返し行う。

【０１２５】図３６は図３５における「マーククリア」の処理内容を示すフローチャートである。この処理はマークテーブルの内容を一旦クリアするものであり、まずマークテーブルの先頭にポインタを移動し（ｓ１３１）、その位置のマークをクリアし（ｓ１３２）、次のマークの位置までポインタを移動させる（ｓ１３３）。この処理を全てのマークについて繰り返す（ｓ１３４→ｓ１３２→・・・）。

【０１２６】図３７は図３５における「オブジェクトの消去」の処理内容を示すフローチャートである。先ず、マークテーブルの先頭にポインタを移動させ（ｓ１４１）、マークの有無を検出し、マークされていなければ、そのマークテーブル上の位置に相当するオブジェクトのヒープ領域内の位置を計算し、該当のオブジェクトを消去する（ｓ１４２→ｓ１４３→ｓ１４４）。続いて、マークテーブルのポインタを次に移動して、同様の処理を繰り返す（ｓ１４５→ｓ１４６→ｓ１４２→・・・）。これによりマークテーブルにマークされているオブジェクトを残し、その他のオブジェクトをヒープ領域から消去する。

【０１２７】説明を容易にするために、先ず前提となるマーク＆スイープ法におけるマーク付与について説明する。

【０１２８】図３８はツリー探索によるマーク付与の手順を示す図である。（Ａ）に示すように、ルートノード１０から各ノードへ、ツリー構造で表される参照関係を辿って、参照関係にあるノード（オブジェクト）にマークを付与する。具体的にはマークテーブル上の該当位置のビットをセットする。このツリー構造は、たとえば或るオブジェクトが他のどのオブジェクトを参照しているかを示す、オブジェクト内に設けられる変数の内容によって構成され、このオブジェクトの参照関係を辿ることが、すなわちツリーを辿ることである。

【０１２９】（Ａ）のように、ノード番号３までマーク付与を行った時点で割込が発生し、その割込処理によって、（Ｂ）のように、ルートノードからノード番号７で表されるオブジェクトへの参照関係が絶たれて、ノード番号２で表されるオブジェクトからノード番号７で表されるオブジェクトが参照される関係となれば、割込処理が終了してＧＣスレッドに戻って、マーク付与を再開したとき、ルートノードからノード番号７で表されるオブジェクトへの参照関係が絶たれているので、（Ｃ）に示すようにポインタをルートノードに戻して次の参照関係にあるノード番号８に進むことになる。この時点ではノード番号５，６に対してはマーク付与されない。そこで、参照関係の変更のあったオブジェクトについては、そのオブジェクトからツリーを辿ってそのオブジェクトから参照されているオブジェクトについてマークを付与する必要がある。

【０１３０】図３９は「オブジェクトの生成」の処理内容を示すフローチャートである。まずカーネルに対してシステムのロックを起動し（ｓ１５１）、ヒープ領域内の空いている領域を探し（ｓ１５２）、生成しようとするオブジェクトのサイズより大きな領域に対して必要なサイズを割り当て（ｓ１５３→ｓ１５４）、参照変更のマーク付与（ライトバリア）を行い（ｓ１５５）、システムをアンロックする（ｓ１５６）。

【０１３１】図４０は上記「参照変更のマーク付与」の処理手順を示すフローチャートである。先ず、参照変更されたオブジェクトからマークテーブル上の位置を計算し、該当のマークがWhite であるか否かを判定する。このWhite マークはたとえば00の２ビットで表され、未だマークされていない状態を意味する。もし、White マークでなければ、すでにマークされているので、そのまま処理を終了する。White マークであれば、それをGrayにマークする。このGrayマークはたとえば01の２ビットで表され、参照変更のあったオブジェクトであることを意味する。なお、オブジェクトからマークの位置を計算する際、たとえばオブジェクトのアドレスを１／８してオフセットを加えることによって行うか、オブジェクトの

通し番号により計算する。

【０１３２】図４１は「ツリー探索によるマーク付与」の処理手順を示すフローチャートである。まず、ツリーを辿るためのポインタをツリーのルートノードに移動させ（ｓ１６１）、新たに生成するオブジェクトに対応するマークを付与する（ｓ１６２）。続いてツリーを辿って、ポインタを次のオブジェクトへ移動させ（ｓ１６３）、この処理をツリーの最後まで繰り返す（ｓ１６４→ｓ１６２→・・・）。その後、各スレッドスタックの先頭へポインタを移動させ（ｓ１６５）、スタックにあるオブジェクトに対応するマークを付与する（ｓ１６６）。その後、ポインタをスタックの次に移動させて（ｓ１６７）、この処理をツリーの最後まで繰り返す（ｓ１６８→ｓ１６６→・・・）。その後、スレッドスタックの次に移り（ｓ１６９）、同様の処理をスレッドスタックの最後まで繰り返す（ｓ１７０→ｓ１６６→・・・）。さらにこのスレッドスタックについての処理をすべてのスレッドスタックについて行う（ｓ１７１→ｓ１７２→ｓ１６５→・・・）。この一連のツリー探索において、Grayマークが途中で１度でも検出された場合、もう一度ルートノードから探索およびマーク付与を行う（ｓ１７３→ｓ１６１→・・・）。

【０１３３】図４２は図４１における「オブジェクトに対応するマーク付与」の処理手順を示すフローチャートである。この処理は、生成されているオブジェクトからマークテーブル上の位置を計算し、Black マークを付与する。このBlack マークはたとえば1xの２ビットで表され、マークされている状態を意味する。

【０１３４】上述したようなマーク付与の方法では、その途中で割込がかかってオブジェクトの参照関係が変化すると、Grayマークが付与されるので、ツリー探索を繰り返さなければならない。場合によってはいつまでたってもマーク付与が完了せずに、ＧＣがいつまでも行われないといった事態に陥る。

【０１３５】図４３は上記の問題を解消するための「ツリー探索によるマーク付与」の手順を示す図である。（Ａ）は図３８に示した（Ａ）の状態で割込がかかって参照関係が変化したときの状態を示す。このように、ノード番号３までマーク付与を行った時点で割込が発生し、その割込処理によって、ルートノードからノード番号７で表されるオブジェクトへの参照関係が絶たれて、ノード番号２で表されるオブジェクトからノード番号７で表されるオブジェクトが参照される関係となれば、参照先のノード番号７表すデータをマークスタックに積む。その後、割込処理が終了してＧＣスレッドに戻って、マーク付与を再開したとき、ルートノードからノード番号７で表されるオブジェクトへの参照関係が絶たれているので、（Ｂ）に示すようにポインタをルートノードに戻して次の参照関係にあるノード番号８をマークする。この時点で一通りのツリー探索を終了し、その後は、（Ｃ）に示すようにマークスタックの内容によって示されるノードからツリーを辿って参照関係にあるオブジェクトについてマークを付与する。これにより（Ｄ）に示すように、参照関係にある全てのオブジェクトについてのマーク付与が完了する。

【０１３６】図４４は「参照変更のマーク付与」の処理手順を示すフローチャートである。このように、参照変更されたオブジェクトを示すデータを上記マークスタックに積む。なお、オブジェクトの生成の処理は図３９に示したもの変わらない。

【０１３７】図４５は、「ツリー探索によるマーク付与」の処理手順を示すフローチャートである。ステップｓ１６１～ｓ１７２の部分は図４１に示したフローチャートのステップｓ１６１～ｓ１７２と同じである。この図４５に示す例では、全スレッドスタックについてのツリー探索を完了した後、マークスタックからデータを取り出し（ｓ１８１→ｓ１８２）、そのデータで示されるオブジェクトに対応するマーク付与を行い（ｓ１８３）、そのオブジェクトからツリーを辿ってツリーの最後まで、参照関係にあるオブジェクトのマーク付与を行う（ｓ１８４→ｓ１８５→ｓ１８３→・・・）。この処理をマークスタックが空になるまでマークスタックのポインタを更新しながら繰り返す（ｓ１８１→ｓ１８２→・・・）。

【０１３８】図４６は図４５における「オブジェクトに対応するマーク付与」の処理手順を示すフローチャートである。この処理は、生成されているオブジェクトからマークテーブル上の位置を計算し、マークを付与する。

【０１３９】このようにマークスタックを用いることによって、ツリー探索をルートノードから再開する必要がなくなり、マーク付与に要する全体の処理時間を大幅に短縮化できる。

【０１４０】図４７および図４８は上記マークスタックを用いてマーク付与を行う場合に、更に無駄な処理時間を無くすようするフローチャートである。図４７は上記「参照変更のマーク付与」の処理手順を示すフローチャートである。先ず、参照変更されたオブジェクトからマークテーブル上の位置を計算し、該当のマークがWhiteであるか否かを判定する（ｓ１９１→ｓ１９２）。このWhite マークは上述したように未だマークされていない状態を意味する。もし、White マークでなければ、すでにマークされているので、そのまま処理を終了する。White マークであれば、それをGrayにマークする（ｓ１９３）。上述したようにこのGrayマークは参照変更のあったオブジェクトであることを意味する。続いてマークスタックに参照先のオブジェクトを示すデータを積む（ｓ１９４）。

【０１４１】図４８は上記「オブジェクトに対応するマーク付与」の処理手順を示すフローチャートである。この処理は、生成されているオブジェクトからマークテー

ブル上の位置を計算し、Black マークを付与する。このBlack マークは上述したように、マークされている状態を意味する。なお、「オブジェクトの生成」の処理は図３９に示したものと同様である。

【０１４２】このように、参照変更のあったオブジェクトについてマークを付与する場合、そのオブジェクトが初めて検出されたオブジェクトである場合にのみマークを付与するようにしたため、マークスタックの内容によるツリー探索に要する時間およびマークスタックの読み出しに要する時間を短縮化できる。

【０１４３】なお、参照変更のあったオブジェクトについてのマークを記憶するのはスタックでなくてもよく、ＦＩＦＯのバッファを用いてもよい。

【０１４４】また、実施形態ではマークテーブルにマークを付与するようにしたが、オブジェクトの内部にマーク用の情報を設けて、オブジェクトに直接マークを付与するようにしてもよい。

【０１４５】

【発明の効果】請求項１，４，５に係る発明によれば、或るスレッドからのコンテキストスイッチ発生有無検出の開始を依頼するＡＰＩの呼び出しからコンテキストスイッチ発生有無検出の終了を依頼するＡＰＩの呼び出しまでの間で行われたスレッドの処理の途中で、コンテキストスイッチがあったか否かが、そのスレッドで分かるため、コンピュータ資源をロックのメカニズムとして使用せずに、処理の排他性が保証される。

【０１４６】また、コンテキストスイッチが発生していれば、その間のスレッドの処理を無効にして、例えばその処理を再度実行するなどの方法によって高い応答性を保ちながら、排他制御を行うことができるようになる。

【０１４７】さらに、或るスレッドの優先度を高い状態と低い状態とに交互に変更するようにしておき、スレッドの優先度が高い状態のとき、該スレッドの優先度が低い状態になるまでの残時間が、必要な処理時間より短いとき、その処理が行われることがないため、その分のＣＰＵパワーを無駄にすることがなく、全体の処理を効率よく進められる。

【０１４８】請求項２に係る発明によれば、システムをロックすることなく、ＧＣを開始することができるので、リアルタイム性を保証することができる。

【０１４９】請求項３に係る発明によれば、システムをロックすることなく、メモリコンパクションを開始することができるので、リアルタイム性を保証することができる。

【０１５０】
【０１５１】
【０１５２】
【０１５３】
【０１５４】
【０１５５】
【０１５６】
【０１５７】
【０１５８】
【０１５９】
【０１６０】
【０１６１】
【０１６２】
【０１６３】

【図面の簡単な説明】

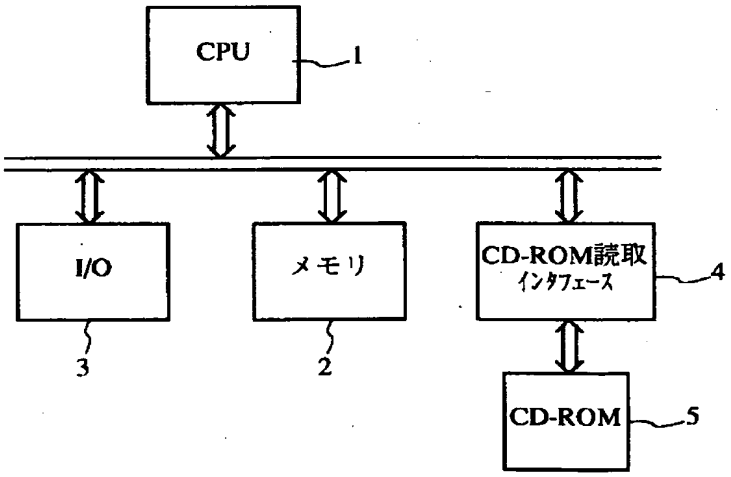
【図１】実施形態に係る装置のハードウエアの構成を示すブロック図
【図２】同装置のソフトウエアの構成を示すブロック図
【図３】オブジェクト間の参照関係を示すツリーおよび各スレッドのスタックとの関係を示す図
【図４】ソフトウエアの機能ブロック図
【図５】コンパクションの作用を説明するための図
【図６】コンテキストスイッチの有無によるスレッドの処理内容の変化の例を示す図
【図７】排他制御の処理手順を示すフローチャート
【図８】コンテキストスイッチ発生有無検出のＡＰＩに関する処理手順を示すフローチャート
【図９】コンテキストスイッチの処理手順を示すフローチャート
【図１０】コンパクションの処理手順を示すフローチャート
【図１１】複写法によるＧＣの処理手順を示すフローチャート
【図１２】複写法ＧＣにおける排他制御の処理手順を示すフローチャート
【図１３】複写法ＧＣにおける他の排他制御の処理手順を示すフローチャート
【図１４】図１３における排他制御用ＡＰＩに関する処理手順を示すフローチャート
【図１５】図１３における排他制御用ＡＰＩに関する処理手順を示すフローチャート
【図１６】ＧＣスレッドの優先度の自動切替の例を示す図
【図１７】スレッドの優先度値および優先度時間の切替に関するフローチャート
【図１８】ＧＣスレッドの高優先度時間を変更した例を示す図
【図１９】ＧＣスレッドの高優先度時間を変更可能とするためのフローチャート
【図２０】ＧＣスレッドの高優先度時間を自動変更する例を示すフローチャート
【図２１】ＧＣスレッドの高・低優先度時間の切り替え周期を変更した例を示す図
【図２２】ＧＣスレッドの高・低優先度時間の切り替え周期を変更するためのフローチャート
【図２３】不定期なＧＣ処理の例を示す図

【図２４】図２１に対応するフローチャート
【図２５】排他制御ＡＰＩによるＧＣスレッドの優先度の強制変更の例を示す図
【図２６】排他制御ＡＰＩによるＧＣスレッドの優先度を強制変更するためのフローチャート
【図２７】ＧＣのアルゴリズムを切り替える処理手順を示すフローチャート
【図２８】生成されるオブジェクトのサイズの分布の例を示す図
【図２９】オブジェクトのメモリの割り当てサイズの例を示す図
【図３０】オブジェクト生成の処理手順を示すフローチャート
【図３１】オブジェクトサイズの分布検知および固定サイズの決定処理に関するフローチャート
【図３２】ヒープ領域の構成を示す図
【図３３】オブジェクト消去の手順を示すフローチャート
【図３４】オブジェクト生成の手順を示すフローチャート
【図３５】マーク＆スイープ法によるＧＣの手順を示すフローチャート
【図３６】マーククリアの処理手順を示すフローチャート
【図３７】オブジェクト消去の処理手順を示すフローチャート

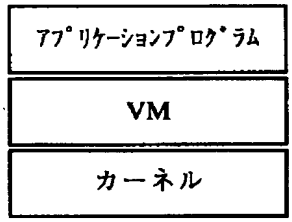
＊【図３８】ツリー探索によるマーク付与の例を示す図
【図３９】図３８に対応するフローチャート
【図４０】図３８に対応するフローチャート
【図４１】図３８に対応するフローチャート
【図４２】図３８に対応するフローチャート
【図４３】ツリー探索によるマーク付与の例を示す図
【図４４】図４３に対応するフローチャート
【図４５】図４３に対応するフローチャート
【図４６】図４３に対応するフローチャート
【図４７】参照変更のマーク付与の他の処理手順を示すフローチャート
【図４８】オブジェクトに対応するマーク付与の他の処理手順を示すフローチャート
【図４９】固定サイズの設定とオブジェクト生成に関する処理手順を示すフローチャート
【図５０】オブジェクトサイズ分布設定に関する処理手順を示すフローチャート
【図５１】オブジェクトサイズの分布と分割サイズの例を示す図
【図５２】ヒープ領域を所定サイズで分割する処理とオブジェクト生成に関する処理手順を示すフローチャート
【図５３】ヒープ領域を所定サイズで分割する処理に関する処理手順を示すフローチャート
【図５４】従来のマーク＆スイープ法によるＧＣの手順を示すフローチャート

＊

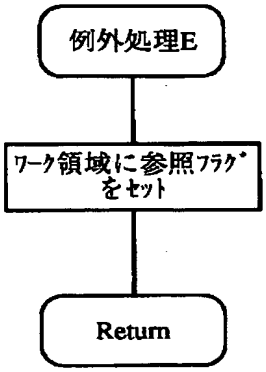
【図１】



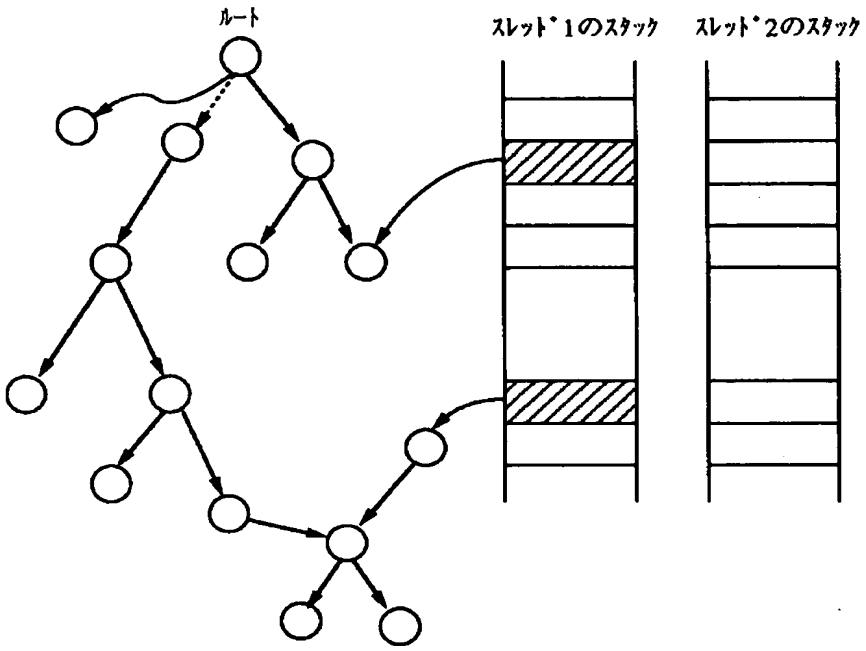
【図２】



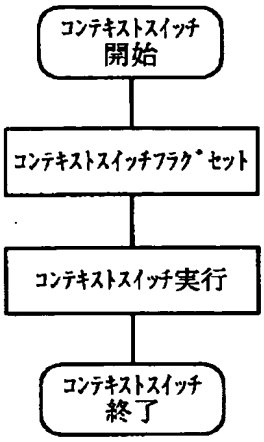
【図１５】



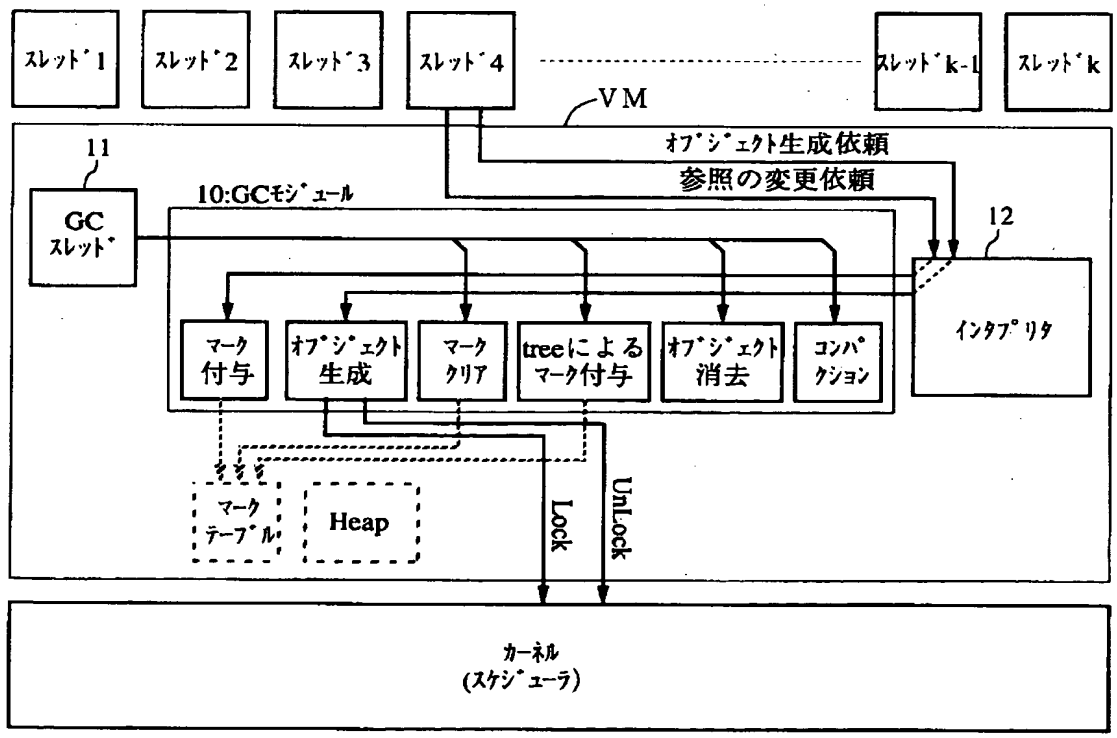
【図３】


ルート
ｽﾚｯﾄﾞ1のｽﾀｯｸ
ｽﾚｯﾄﾞ2のｽﾀｯｸ

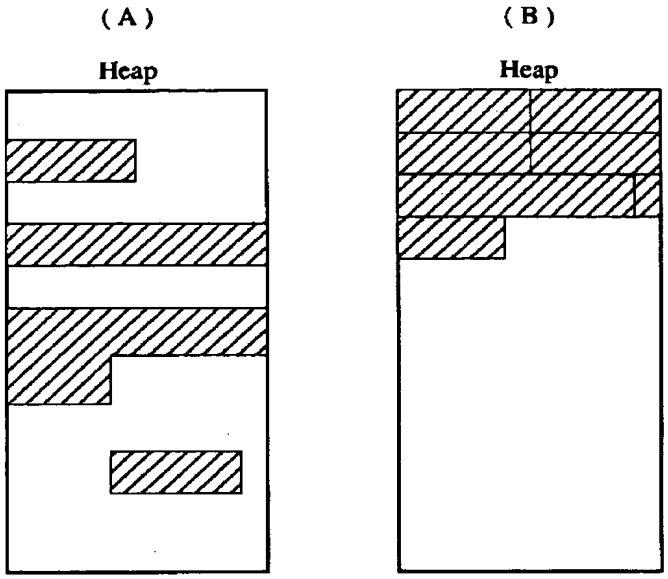
【図９】


ｺﾝﾃｷｽﾄｽｲｯﾁ
開始
ｺﾝﾃｷｽﾄｽｲｯﾁﾌﾗｸﾞｾｯﾄ
ｺﾝﾃｷｽﾄｽｲｯﾁ実行
ｺﾝﾃｷｽﾄｽｲｯﾁ
終了

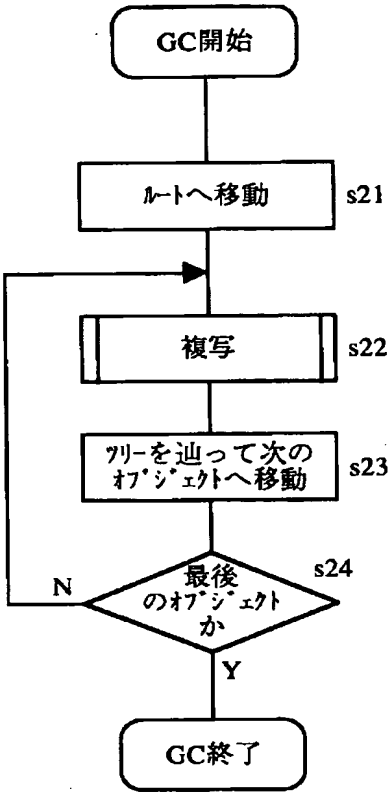
【図４】


ｽﾚｯﾄﾞ1
ｽﾚｯﾄﾞ2
ｽﾚｯﾄﾞ3
ｽﾚｯﾄﾞ4
ｽﾚｯﾄﾞk-1
ｽﾚｯﾄﾞk
VM
11
ｵﾌﾞｼﾞｪｸﾄ生成依頼
参照の変更依頼
10:GCﾓｼﾞｭｰﾙ
GC
ｽﾚｯﾄﾞ
12
ﾏｰｸ
付与
ｵﾌﾞｼﾞｪｸﾄ
生成
ﾏｰｸ
ｸﾘｱ
treeによる
ﾏｰｸ付与
ｵﾌﾞｼﾞｪｸﾄ
消去
ｺﾝﾊﾟｸｼｮﾝ
ｲﾝﾀﾌﾟﾘﾀ
ﾏｰｸ
ﾃｰﾌﾞﾙ
Heap
Lock
UnLock
ｶｰﾈﾙ
(ｽｹｼﾞｭｰﾗ)

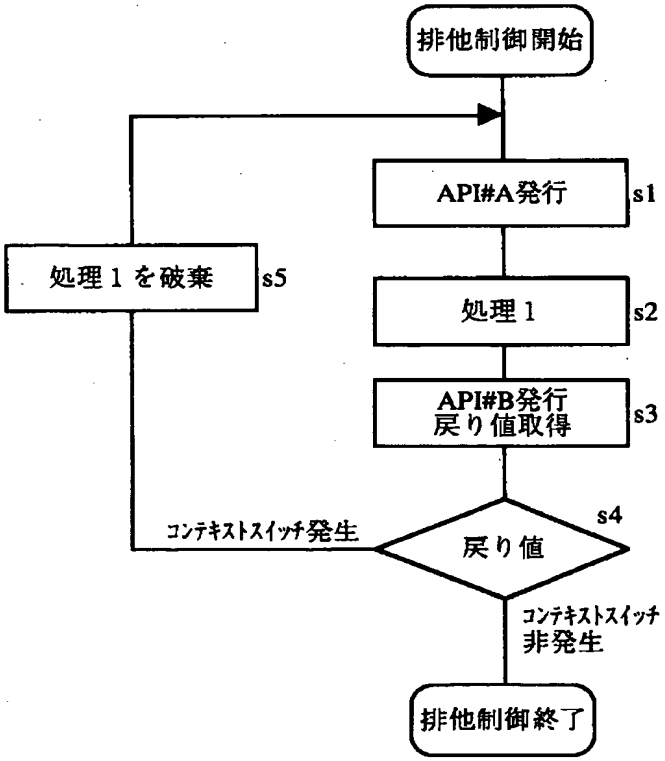
【図５】



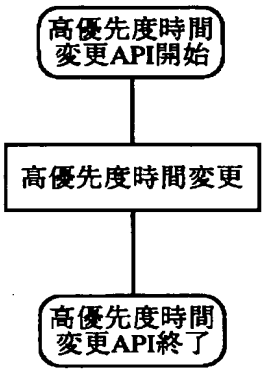
【図１１】



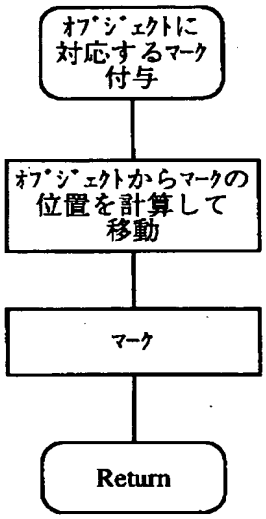
【図７】



【図１９】

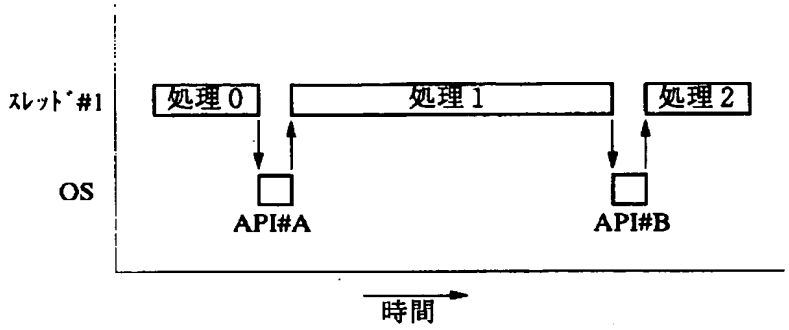


【図４６】

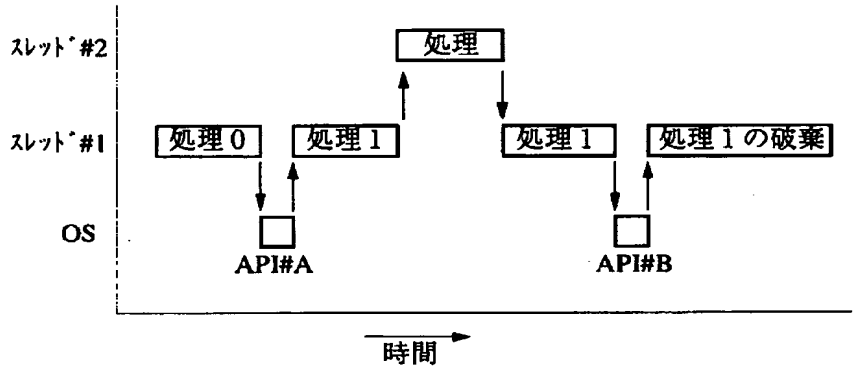


【図６】

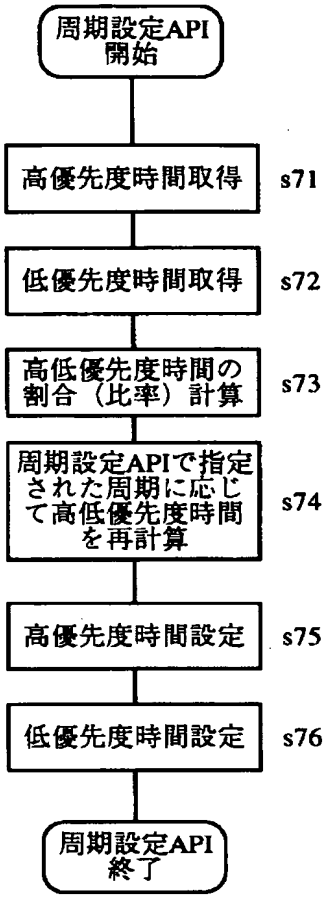
（Ａ）



（Ｂ）

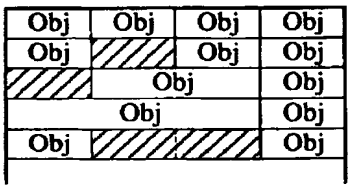


【図２２】

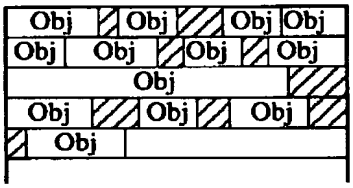


【図２９】

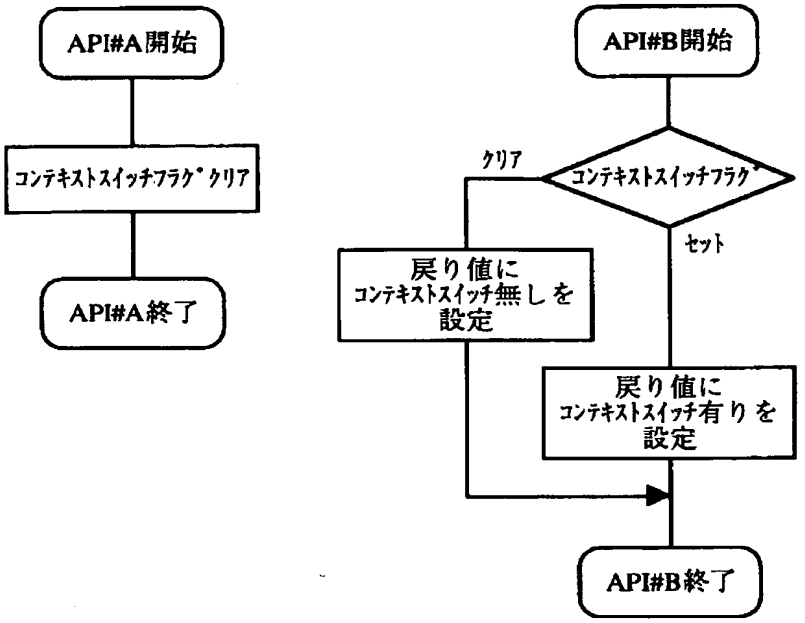
（Ａ）



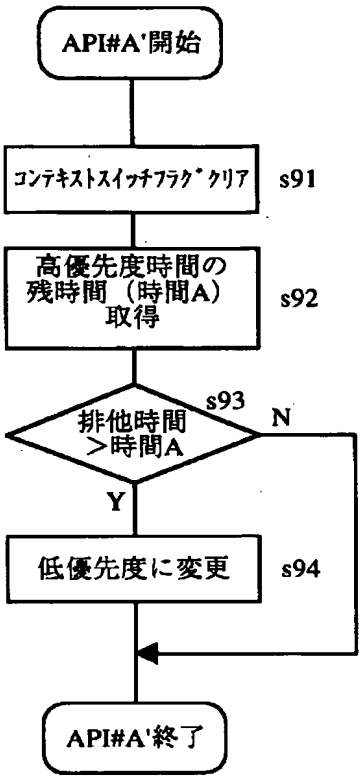
（Ｂ）



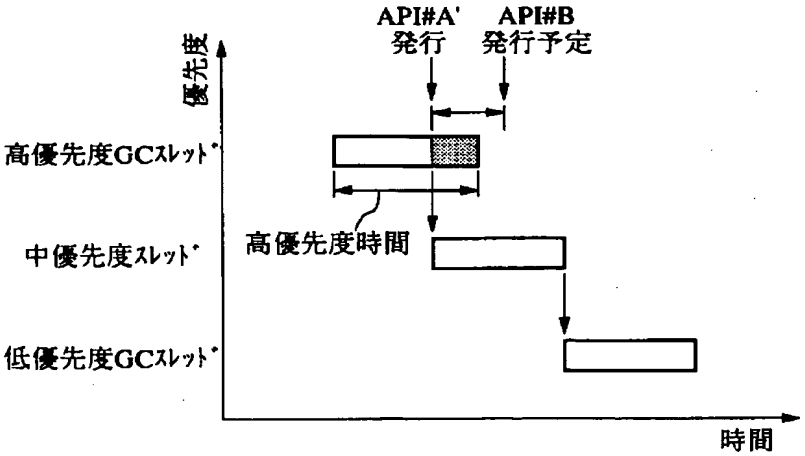
【図８】



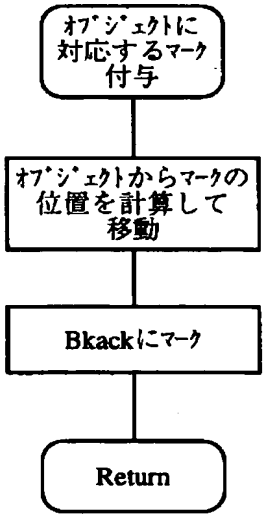
【図２６】



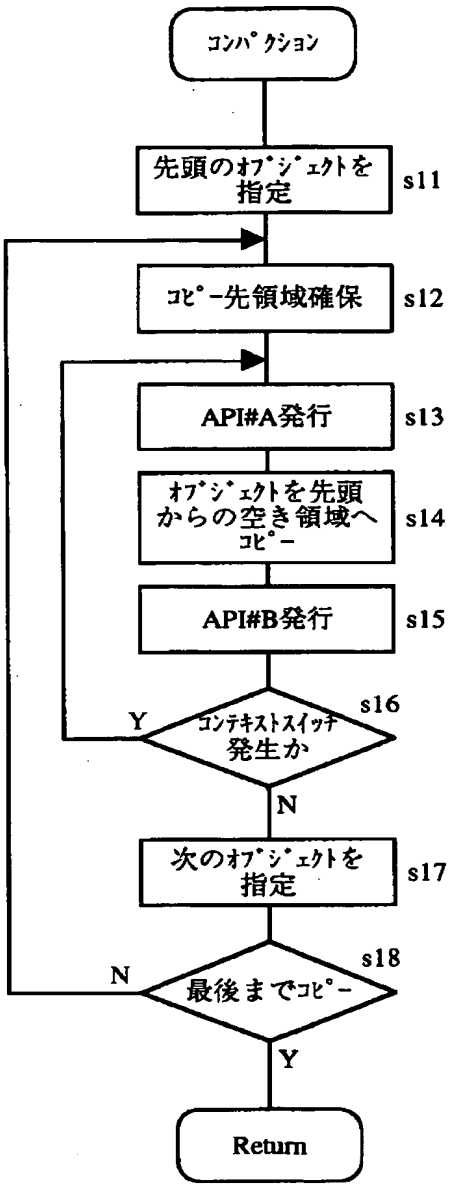
【図２５】



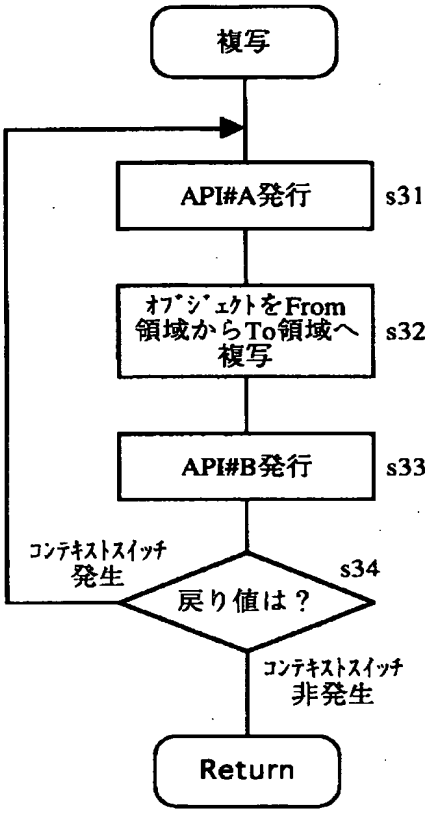
【図４８】



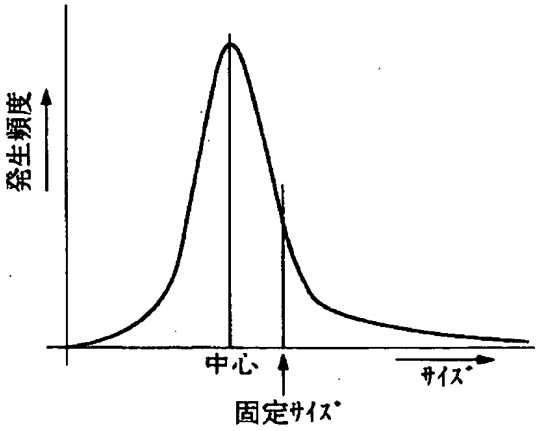
【図１０】


ｺﾝﾊﾟｸｼｮﾝ
先頭のｵﾌﾞｼﾞｪｸﾄを指定
s11
ｺﾋﾟｰ先領域確保
s12
API#A発行
s13
ｵﾌﾞｼﾞｪｸﾄを先頭からの空き領域へｺﾋﾟｰ
s14
API#B発行
s15
ｺﾝﾃｷｽﾄｽｲｯﾁ発生か
s16
Y
N
次のｵﾌﾞｼﾞｪｸﾄを指定
s17
最後までｺﾋﾟｰ
s18
N
Y
Return

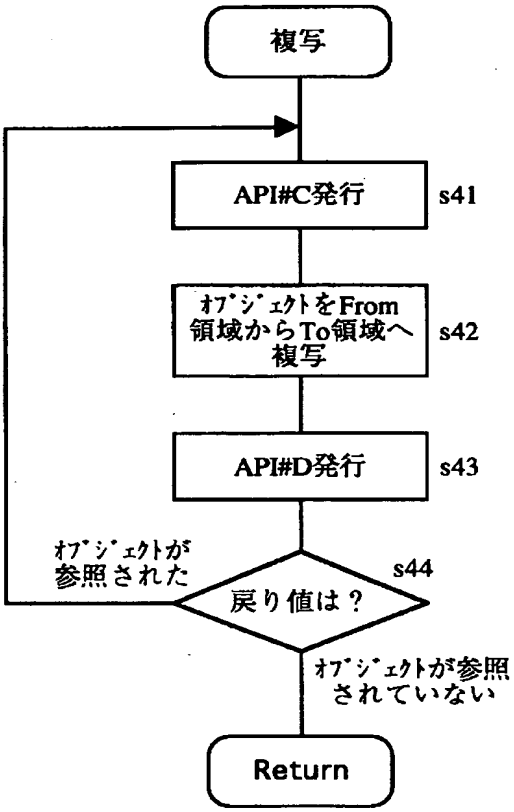
【図１２】


複写
API#A発行
s31
ｵﾌﾞｼﾞｪｸﾄをFrom領域からTo領域へ複写
s32
API#B発行
s33
ｺﾝﾃｷｽﾄｽｲｯﾁ発生
戻り値は？
s34
ｺﾝﾃｷｽﾄｽｲｯﾁ非発生
Return

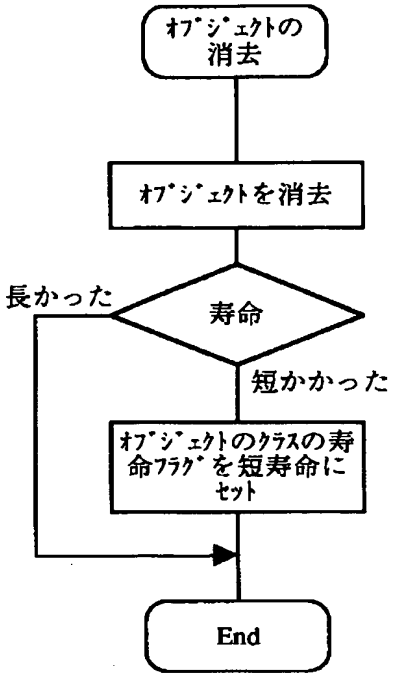
【図２８】


発生頻度
中心
ｻｲｽﾞ
固定ｻｲｽﾞ

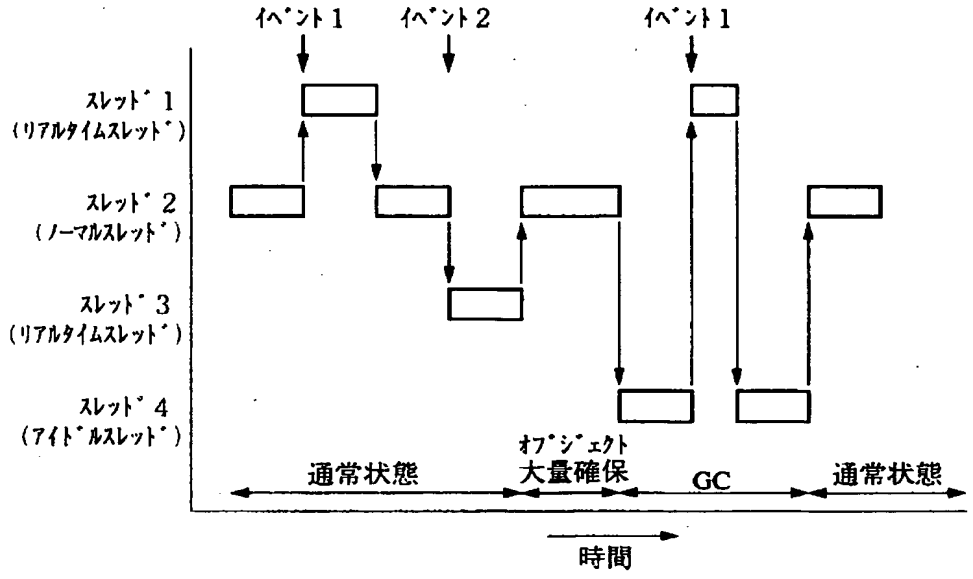
【図１３】


複写
API#C発行
s41
ｵﾌﾞｼﾞｪｸﾄをFrom領域からTo領域へ複写
s42
API#D発行
s43
ｵﾌﾞｼﾞｪｸﾄが参照された
s44
戻り値は？
ｵﾌﾞｼﾞｪｸﾄが参照されていない
Return

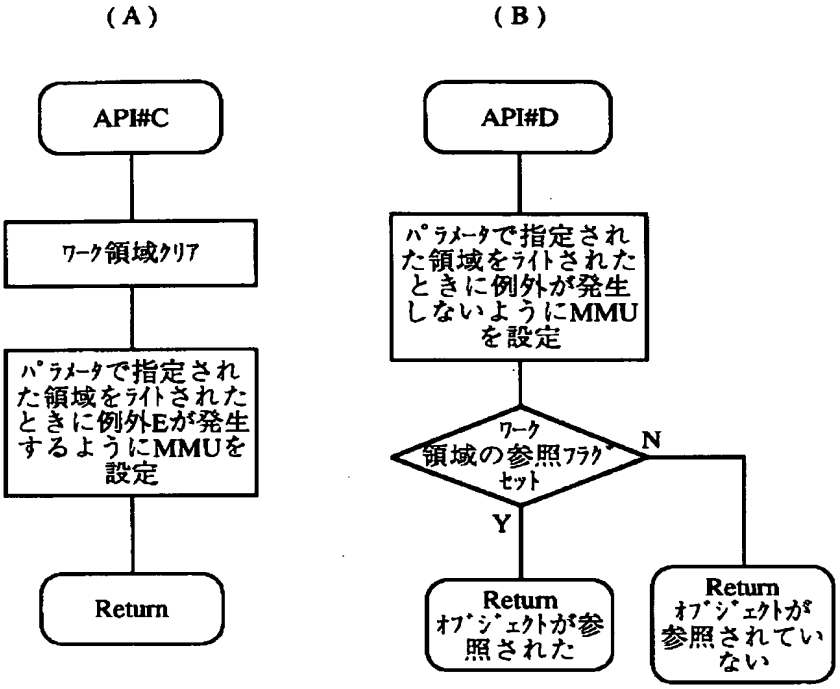
【図３３】


ｵﾌﾞｼﾞｪｸﾄの消去
ｵﾌﾞｼﾞｪｸﾄを消去
長かった
寿命
短かかった
ｵﾌﾞｼﾞｪｸﾄのｸﾗｽの寿命ﾌﾗｸﾞを短寿命にｾｯﾄ
End

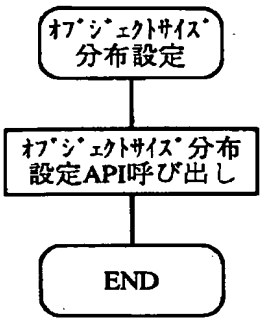
【図２３】


ｲﾍﾞﾝﾄ1
ｲﾍﾞﾝﾄ2
ｲﾍﾞﾝﾄ1
ｽﾚｯﾄﾞ1（ﾘｱﾙﾀｲﾑｽﾚｯﾄﾞ）
ｽﾚｯﾄﾞ2（ﾉｰﾏﾙｽﾚｯﾄﾞ）
ｽﾚｯﾄﾞ3（ﾘｱﾙﾀｲﾑｽﾚｯﾄﾞ）
ｽﾚｯﾄﾞ4（ｱｲﾄﾞﾙｽﾚｯﾄﾞ）
通常状態
ｵﾌﾞｼﾞｪｸﾄ大量確保
GC
通常状態
時間

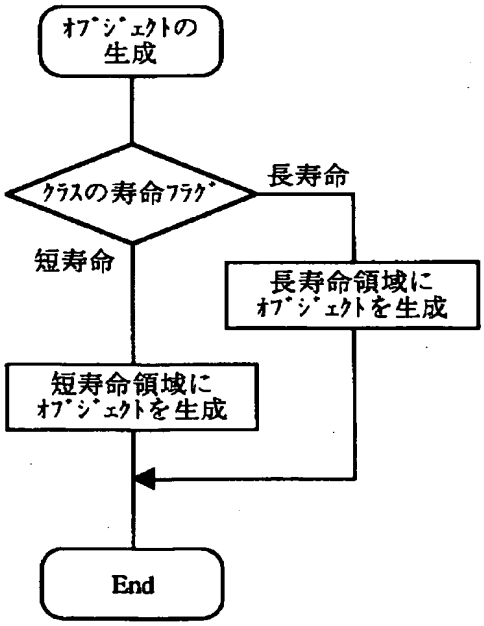
【図１４】

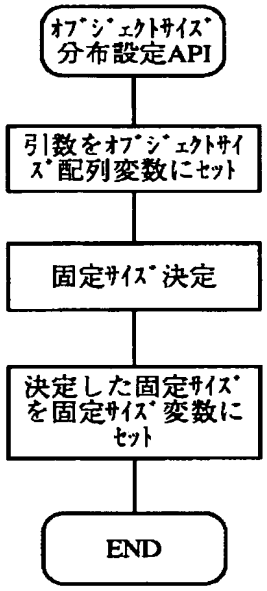

(A)
API#C
ﾜｰｸ領域ｸﾘｱ
ﾊﾟﾗﾒｰﾀで指定された領域をﾗｲﾄされたときに例外Eが発生するようにMMUを設定
Return
(B)
API#D
ﾊﾟﾗﾒｰﾀで指定された領域をﾗｲﾄされたときに例外が発生しないようにMMUを設定
ﾜｰｸ領域の参照ﾌﾗｸﾞｾｯﾄ
N
Y
Return ｵﾌﾞｼﾞｪｸﾄが参照された
Return ｵﾌﾞｼﾞｪｸﾄが参照されていない

【図５０】

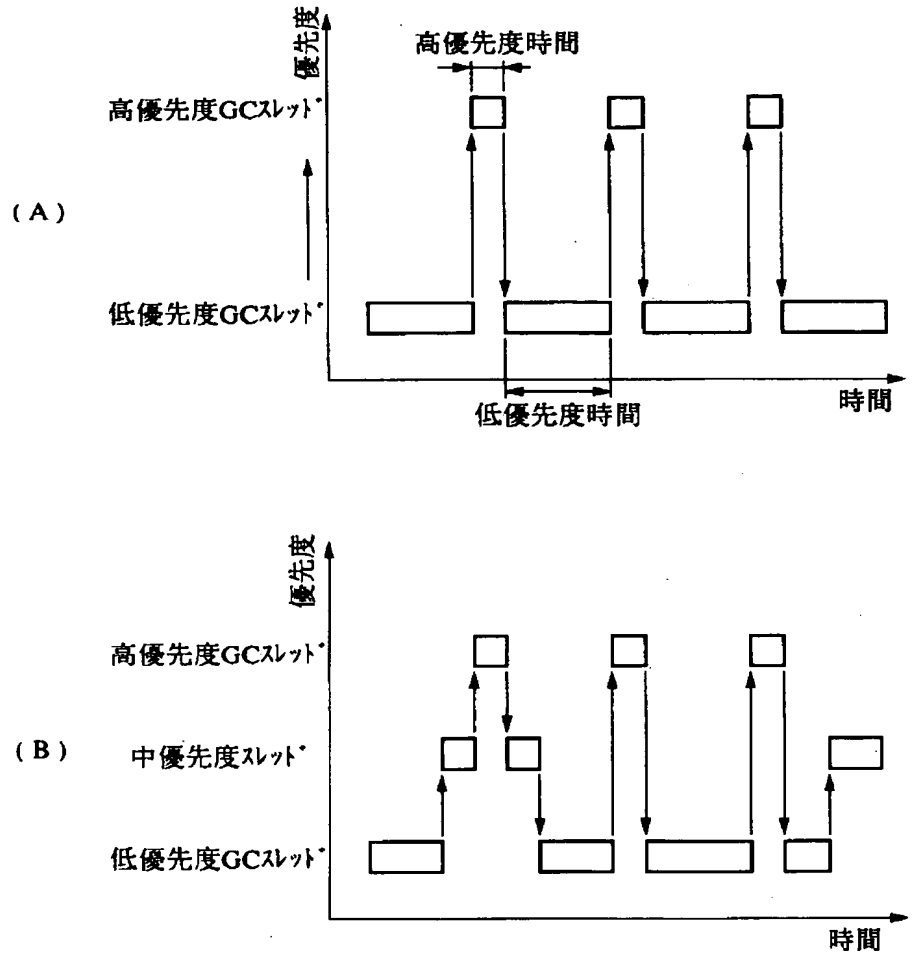

ｵﾌﾞｼﾞｪｸﾄｻｲｽﾞ分布設定
ｵﾌﾞｼﾞｪｸﾄｻｲｽﾞ分布設定API呼び出し
END

【図３４】

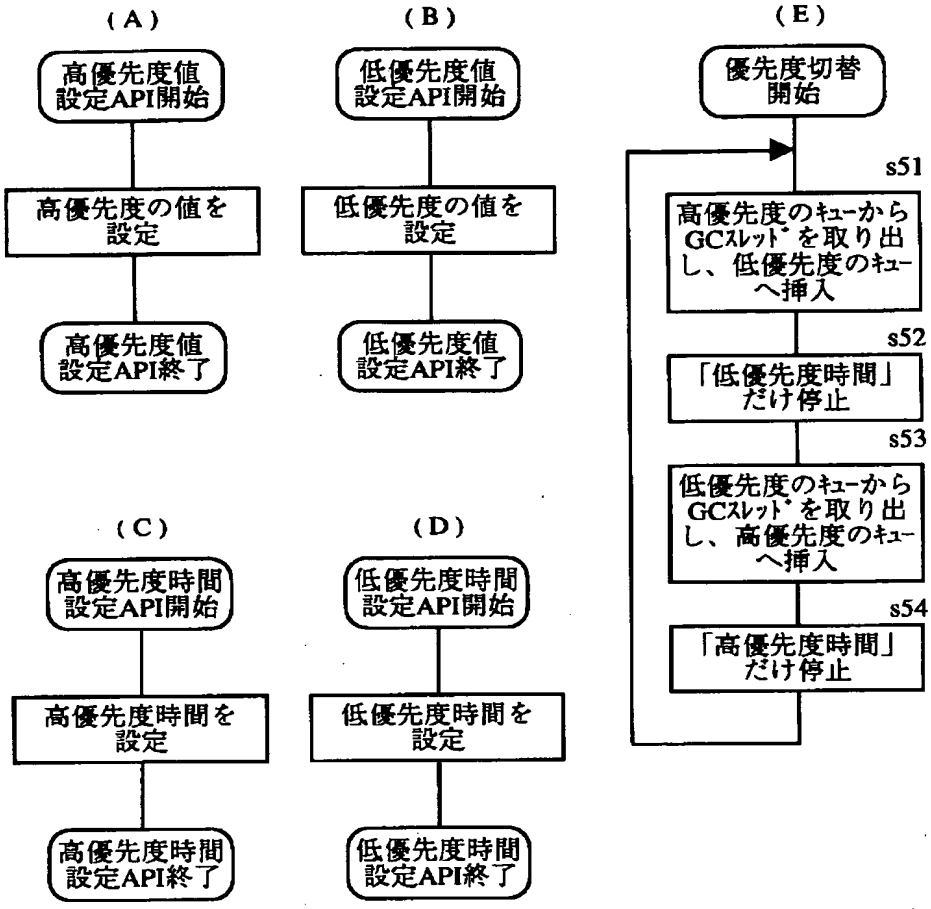

ｵﾌﾞｼﾞｪｸﾄの生成
ｸﾗｽの寿命ﾌﾗｸﾞ
長寿命
短寿命
長寿命領域にｵﾌﾞｼﾞｪｸﾄを生成
短寿命領域にｵﾌﾞｼﾞｪｸﾄを生成
End


ｵﾌﾞｼﾞｪｸﾄｻｲｽﾞ分布設定API
引数をｵﾌﾞｼﾞｪｸﾄｻｲｽﾞ配列変数にｾｯﾄ
固定ｻｲｽﾞ決定
決定した固定ｻｲｽﾞを固定ｻｲｽﾞ変数にｾｯﾄ
END

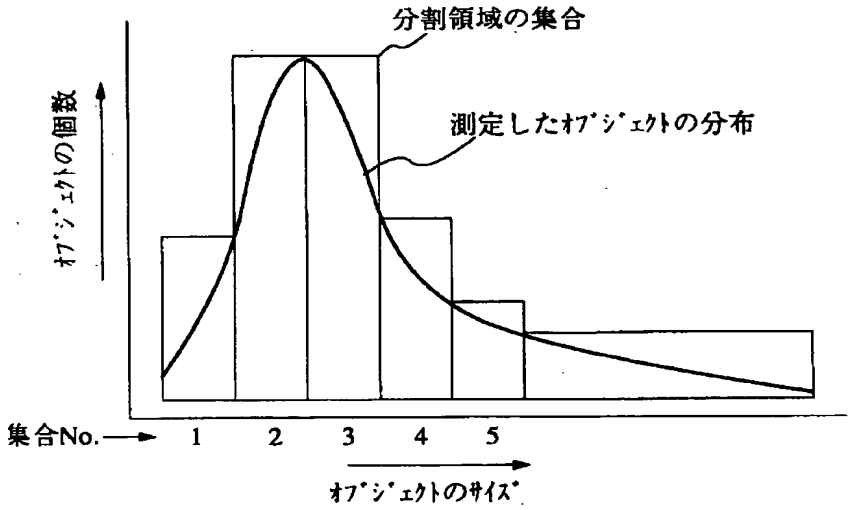
【図１６】


(A)
優先度
高優先度時間
高優先度GCスレッド
低優先度GCスレッド
時間
低優先度時間
(B)
優先度
高優先度GCスレッド
中優先度スレッド
低優先度GCスレッド
時間

【図１７】

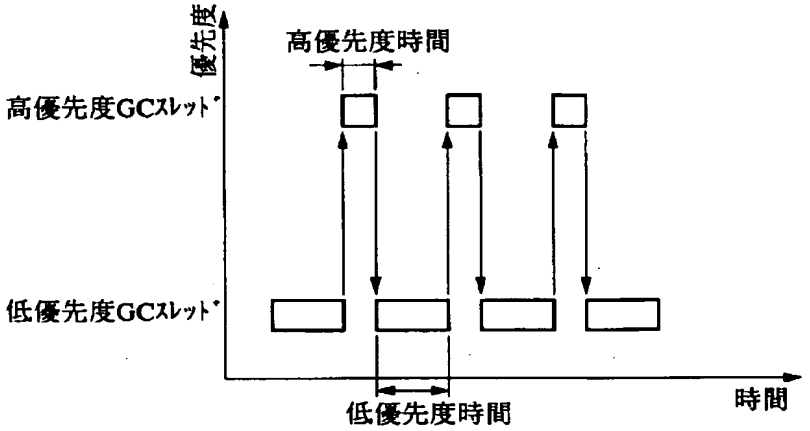

(A)
高優先度値設定API開始
高優先度の値を設定
高優先度値設定API終了
(B)
低優先度値設定API開始
低優先度の値を設定
低優先度値設定API終了
(C)
高優先度時間設定API開始
高優先度時間を設定
高優先度時間設定API終了
(D)
低優先度時間設定API開始
低優先度時間を設定
低優先度時間設定API終了
(E)
優先度切替開始
s51
高優先度のｷｭｰからGCｽﾚｯﾄﾞを取り出し、低優先度のｷｭｰへ挿入
s52
「低優先度時間」だけ停止
s53
低優先度のｷｭｰからGCｽﾚｯﾄﾞを取り出し、高優先度のｷｭｰへ挿入
s54
「高優先度時間」だけ停止

【図５１】

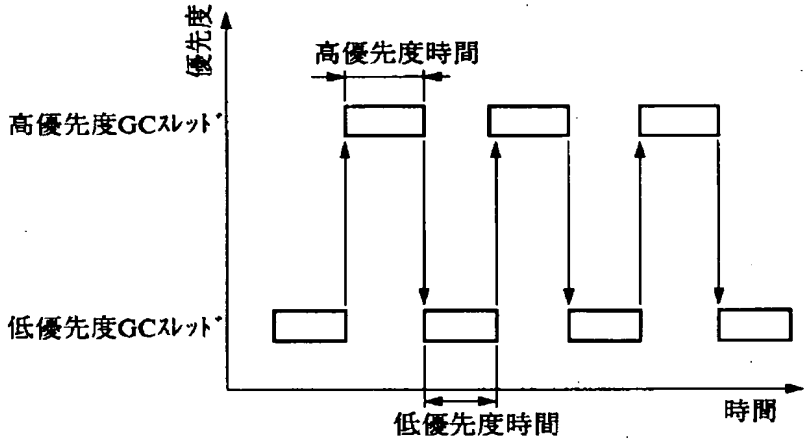

分割領域の集合
測定したｵﾌﾞｼﾞｪｸﾄの分布
ｵﾌﾞｼﾞｪｸﾄの個数
集合No.
1
2
3
4
5
ｵﾌﾞｼﾞｪｸﾄのｻｲｽﾞ

【図１８】

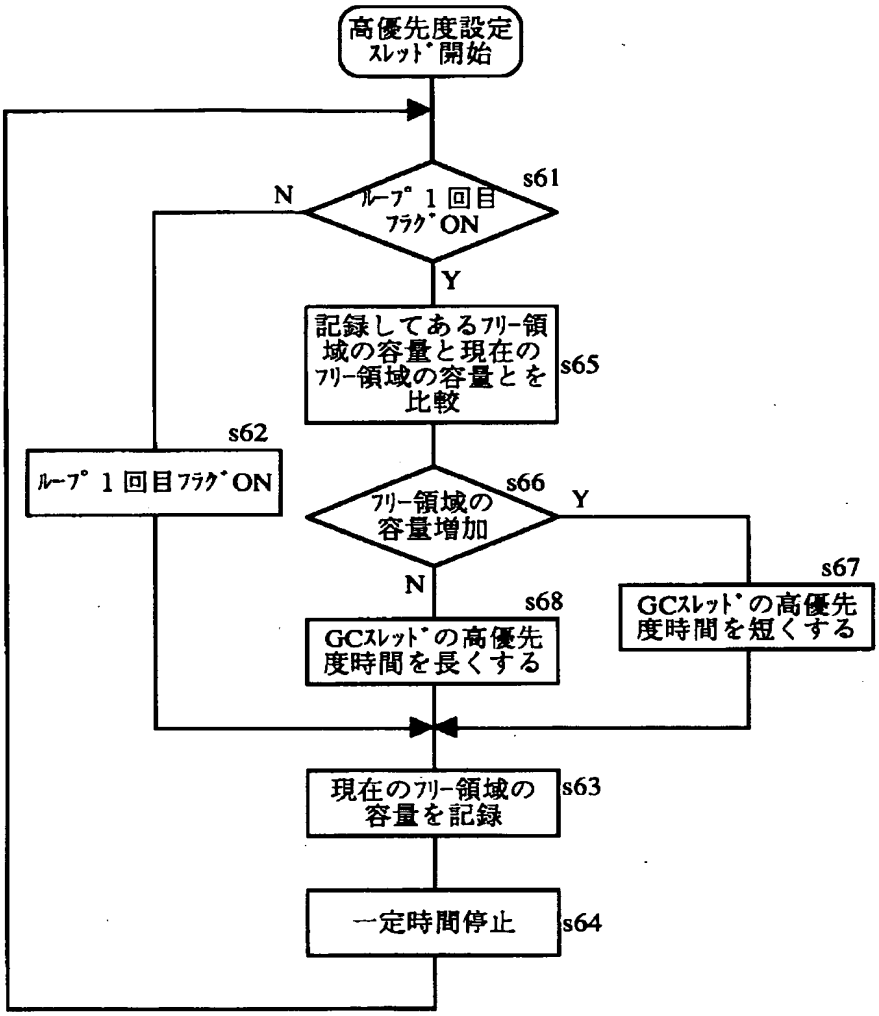
（Ａ）


優先度
高優先度時間
高優先度GCスレッド
低優先度GCスレッド
時間
低優先度時間

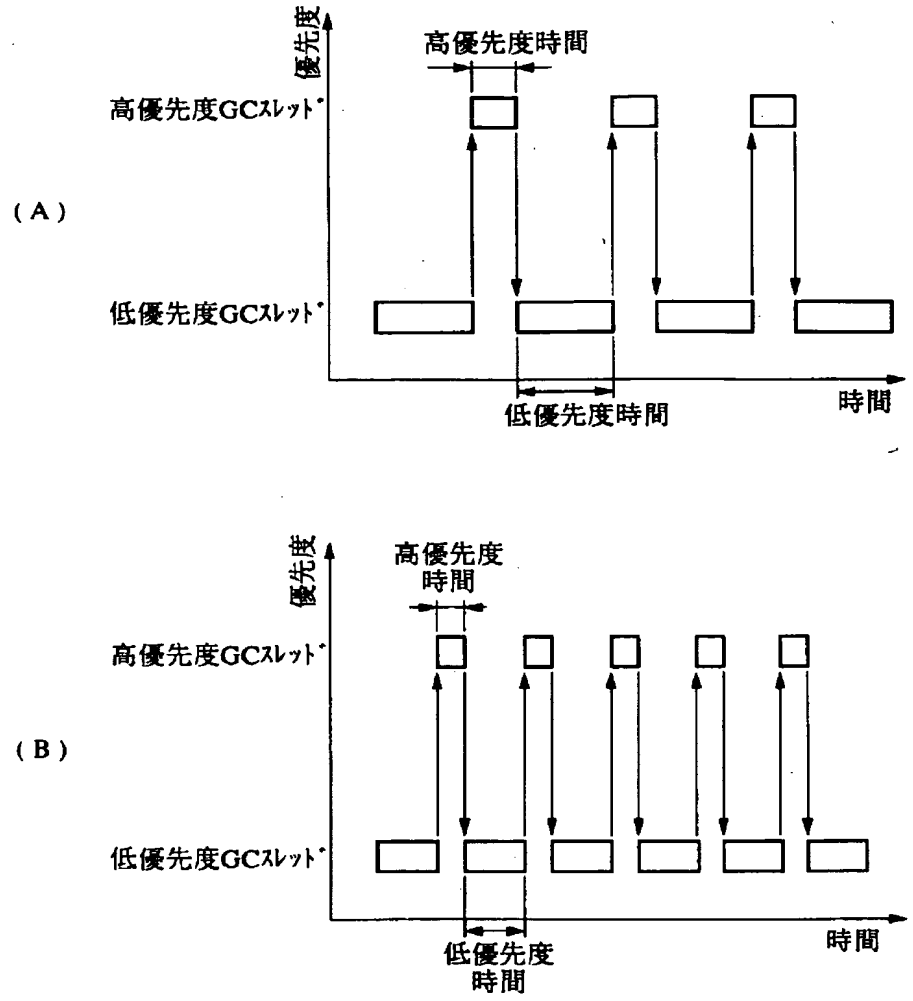
（Ｂ）


優先度
高優先度時間
高優先度GCスレッド
低優先度GCスレッド
時間
低優先度時間

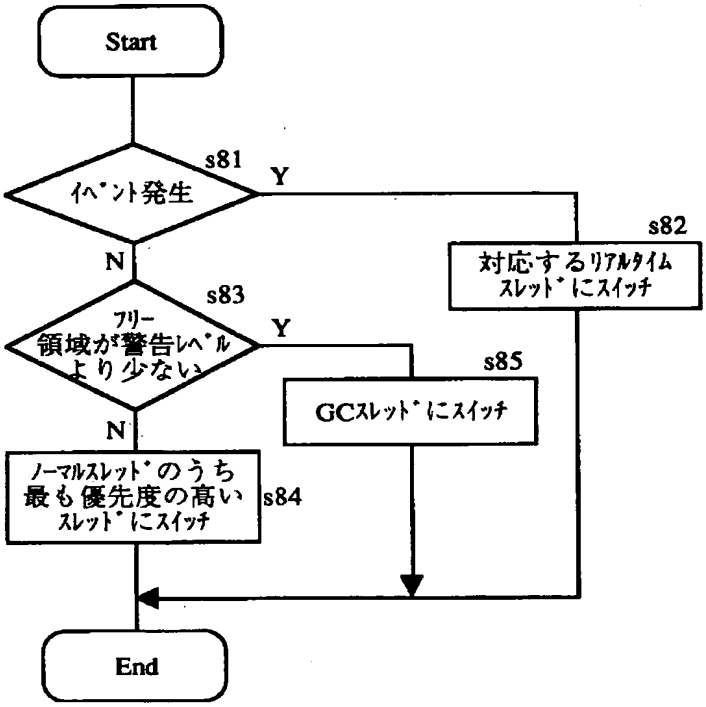
【図２０】


高優先度設定
ｽﾚｯﾄﾞ開始
s61
ﾙｰﾌﾟ１回目
ﾌﾗｸﾞON
N
Y
記録してあるﾌﾘｰ領域の容量と現在のﾌﾘｰ領域の容量とを比較
s65
s62
ﾙｰﾌﾟ１回目ﾌﾗｸﾞON
s66
ﾌﾘｰ領域の容量増加
Y
N
s67
GCｽﾚｯﾄﾞの高優先度時間を短くする
s68
GCｽﾚｯﾄﾞの高優先度時間を長くする
現在のﾌﾘｰ領域の容量を記録
s63
一定時間停止
s64

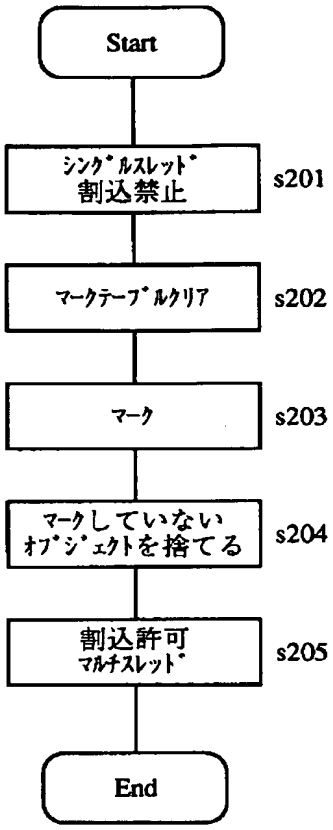
【図２１】


(A)
優先度
高優先度時間
高優先度GCスレッド
低優先度GCスレッド
時間
低優先度時間
(B)
優先度
高優先度
時間
高優先度GCスレッド
低優先度GCスレッド
時間
低優先度
時間

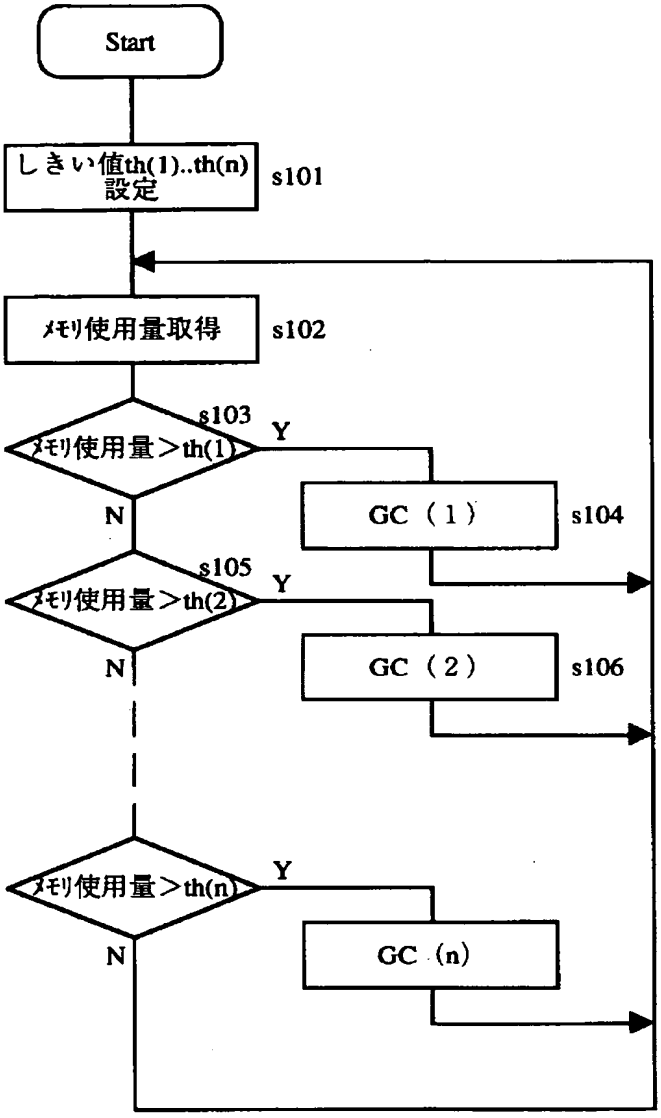
【図２４】


Start
s81
ｲﾍﾞﾝﾄ発生
Y
N
s82
対応するﾘｱﾙﾀｲﾑ
ｽﾚｯﾄﾞにｽｲｯﾁ
s83
ﾌﾘｰ
領域が警告ﾚﾍﾞﾙ
より少ない
Y
s85
GCｽﾚｯﾄﾞにｽｲｯﾁ
N
ﾉｰﾏﾙｽﾚｯﾄﾞのうち
最も優先度の高い
ｽﾚｯﾄﾞにｽｲｯﾁ
s84
End

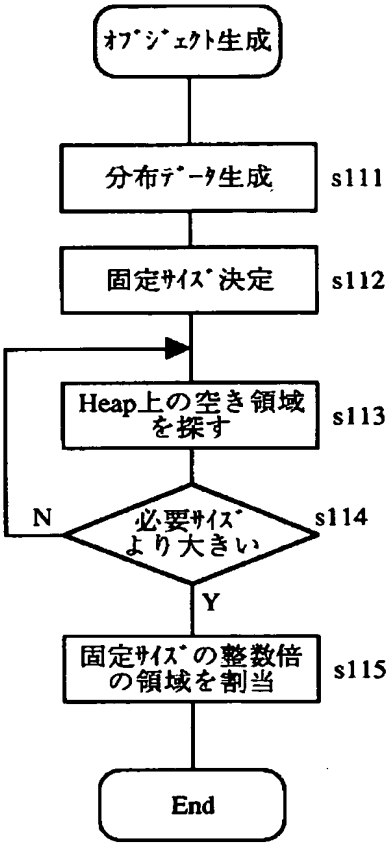
【図５４】


Start
ｼﾝｸﾞﾙｽﾚｯﾄﾞ
割込禁止
s201
ﾏｰｸﾃｰﾌﾞﾙｸﾘｱ
s202
ﾏｰｸ
s203
ﾏｰｸしていない
ｵﾌﾞｼﾞｪｸﾄを捨てる
s204
割込許可
ﾏﾙﾁｽﾚｯﾄﾞ
s205
End

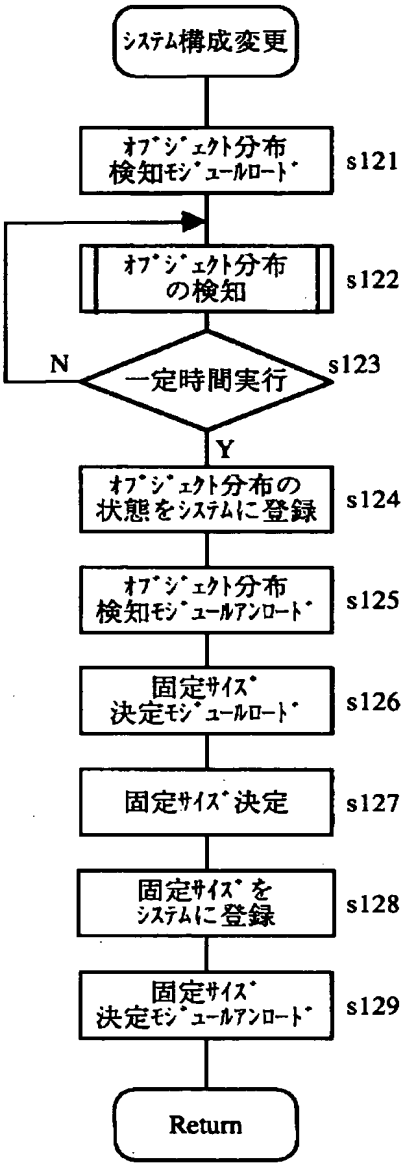
【図２７】


Start
しきい値th(1)..th(n)設定
s101
メモリ使用量取得
s102
s103
メモリ使用量＞th(1)
Y
N
GC（１）
s104
s105
メモリ使用量＞th(2)
Y
N
GC（２）
s106
メモリ使用量＞th(n)
Y
N
GC（n）

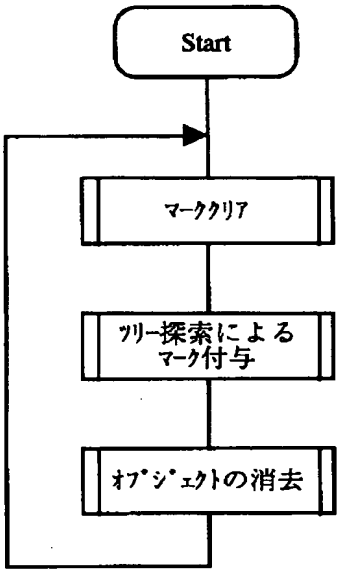
【図３０】


ｵﾌﾞｼﾞｪｸﾄ生成
分布ﾃﾞｰﾀ生成
s111
固定ｻｲｽﾞ決定
s112
Heap上の空き領域を探す
s113
N
必要ｻｲｽﾞより大きい
s114
Y
固定ｻｲｽﾞの整数倍の領域を割当
s115
End

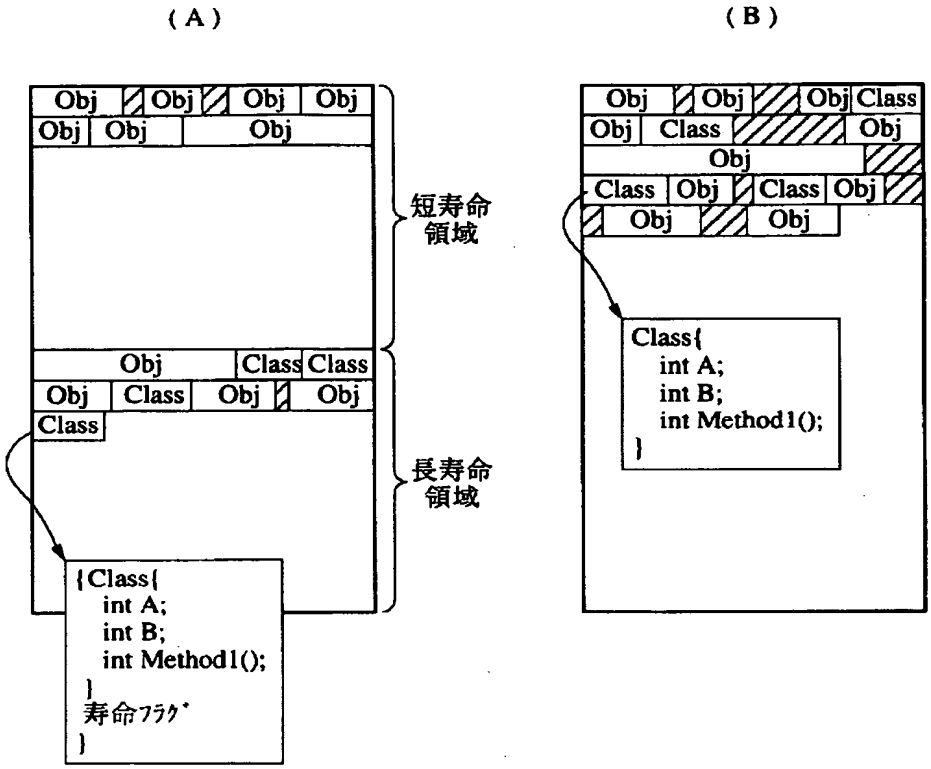
【図３１】


システム構成変更
オブジェクト分布検知モジュールロード
s121
オブジェクト分布の検知
s122
N
一定時間実行
s123
Y
オブジェクト分布の状態をシステムに登録
s124
オブジェクト分布検知モジュールアンロード
s125
固定サイズ決定モジュールロード
s126
固定サイズ決定
s127
固定サイズをシステムに登録
s128
固定サイズ決定モジュールアンロード
s129
Return

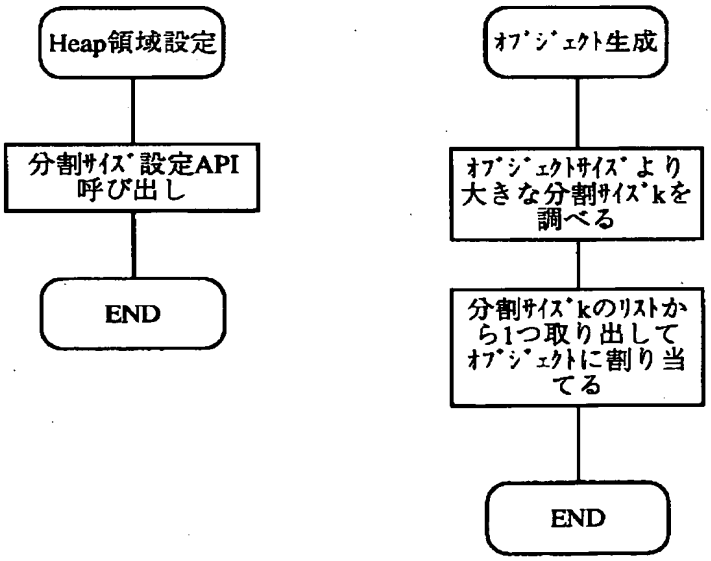
【図３５】


Start
マーククリア
ツリー探索によるマーク付与
オブジェクトの消去

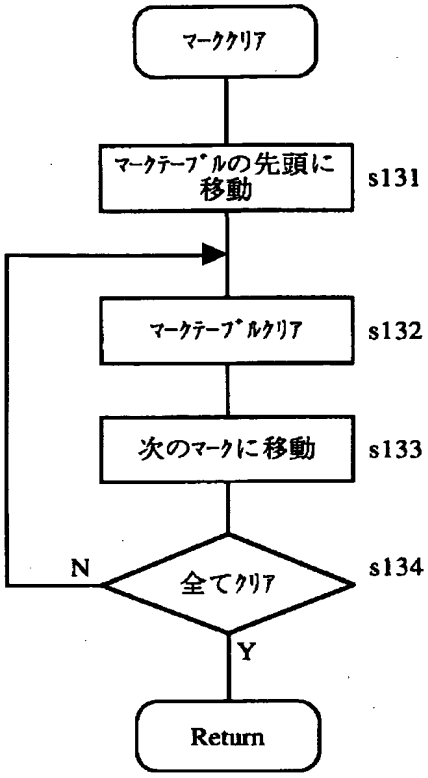
【図３２】


(A)
(B)
Obj
Class
短寿命
領域
長寿命
領域
{Class{
int A;
int B;
int Method1();
}
寿命ﾌﾗｸﾞ
}
Class{
int A;
int B;
int Method1();
}

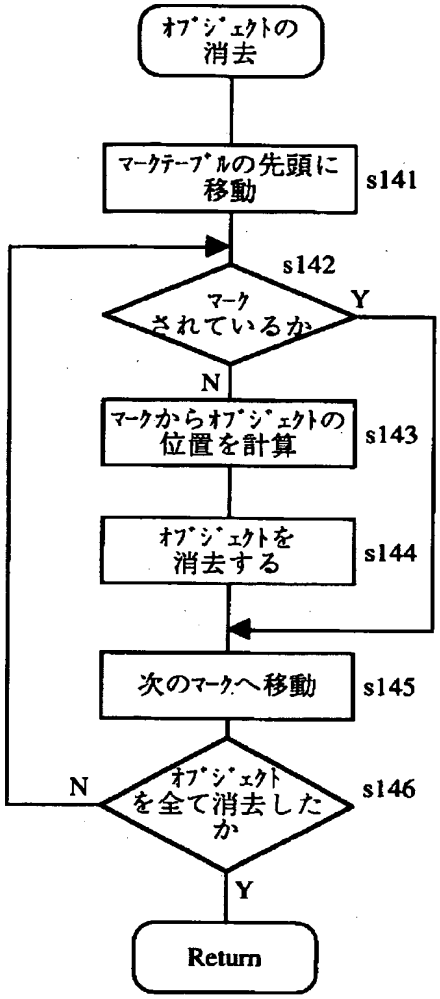
【図５２】


Heap領域設定
分割ｻｲｽﾞ設定API
呼び出し
END
ｵﾌﾞｼﾞｪｸﾄ生成
ｵﾌﾞｼﾞｪｸﾄｻｲｽﾞより
大きな分割ｻｲｽﾞkを
調べる
分割ｻｲｽﾞkのﾘｽﾄから1つ取り出して
ｵﾌﾞｼﾞｪｸﾄに割り当てる
END

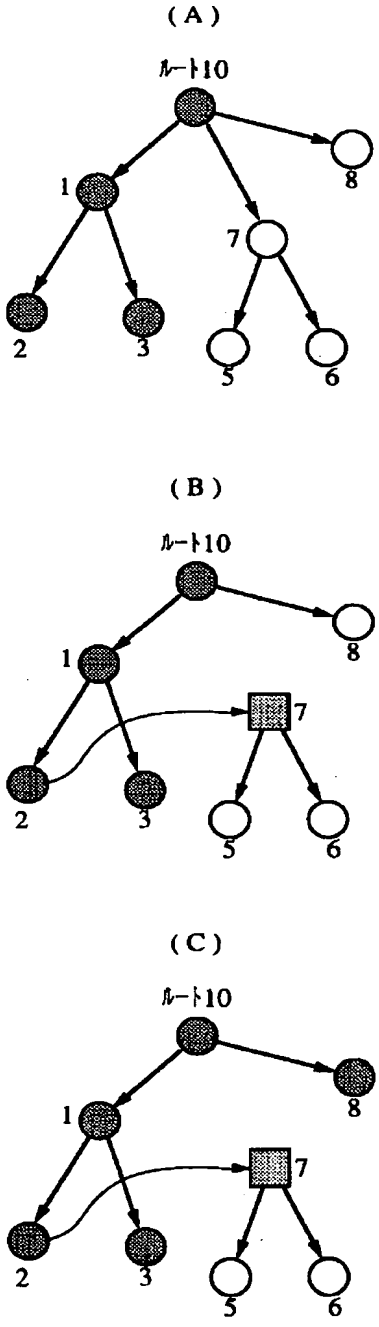
【図３６】


ﾏｰｸｸﾘｱ
ﾏｰｸﾃｰﾌﾞﾙの先頭に移動
s131
ﾏｰｸﾃｰﾌﾞﾙｸﾘｱ
s132
次のﾏｰｸに移動
s133
全てｸﾘｱ
s134
N
Y
Return

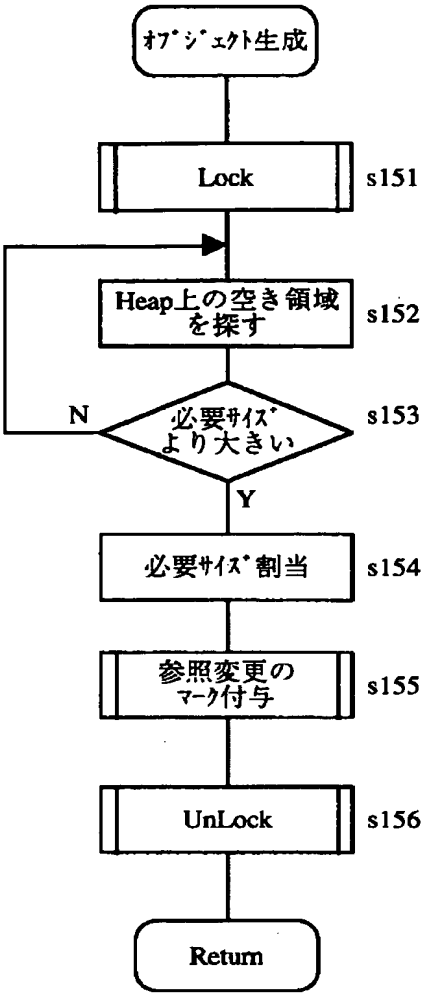
【図３７】


ｵﾌﾞｼﾞｪｸﾄの消去
ﾏｰｸﾃｰﾌﾞﾙの先頭に移動
s141
s142
ﾏｰｸされているか
Y
N
ﾏｰｸからｵﾌﾞｼﾞｪｸﾄの位置を計算
s143
ｵﾌﾞｼﾞｪｸﾄを消去する
s144
次のﾏｰｸへ移動
s145
ｵﾌﾞｼﾞｪｸﾄを全て消去したか
s146
N
Y
Return

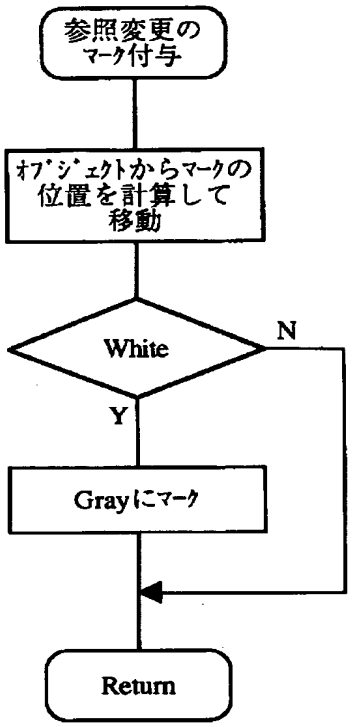
【図３８】


(A)
ルート10
1
8
7
2
3
5
6
(B)
ルート10
1
8
7
2
3
5
6
Black
White
Gray
(C)
ルート10
1
8
7
2
3
5
6

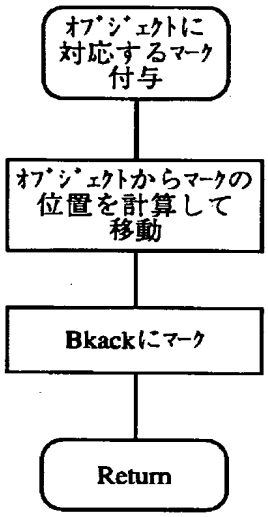
【図３９】


ｵﾌﾞｼﾞｪｸﾄ生成
Lock
s151
Heap上の空き領域を探す
s152
N
必要ｻｲｽﾞより大きい
s153
Y
必要ｻｲｽﾞ割当
s154
参照変更のﾏｰｸ付与
s155
UnLock
s156
Return

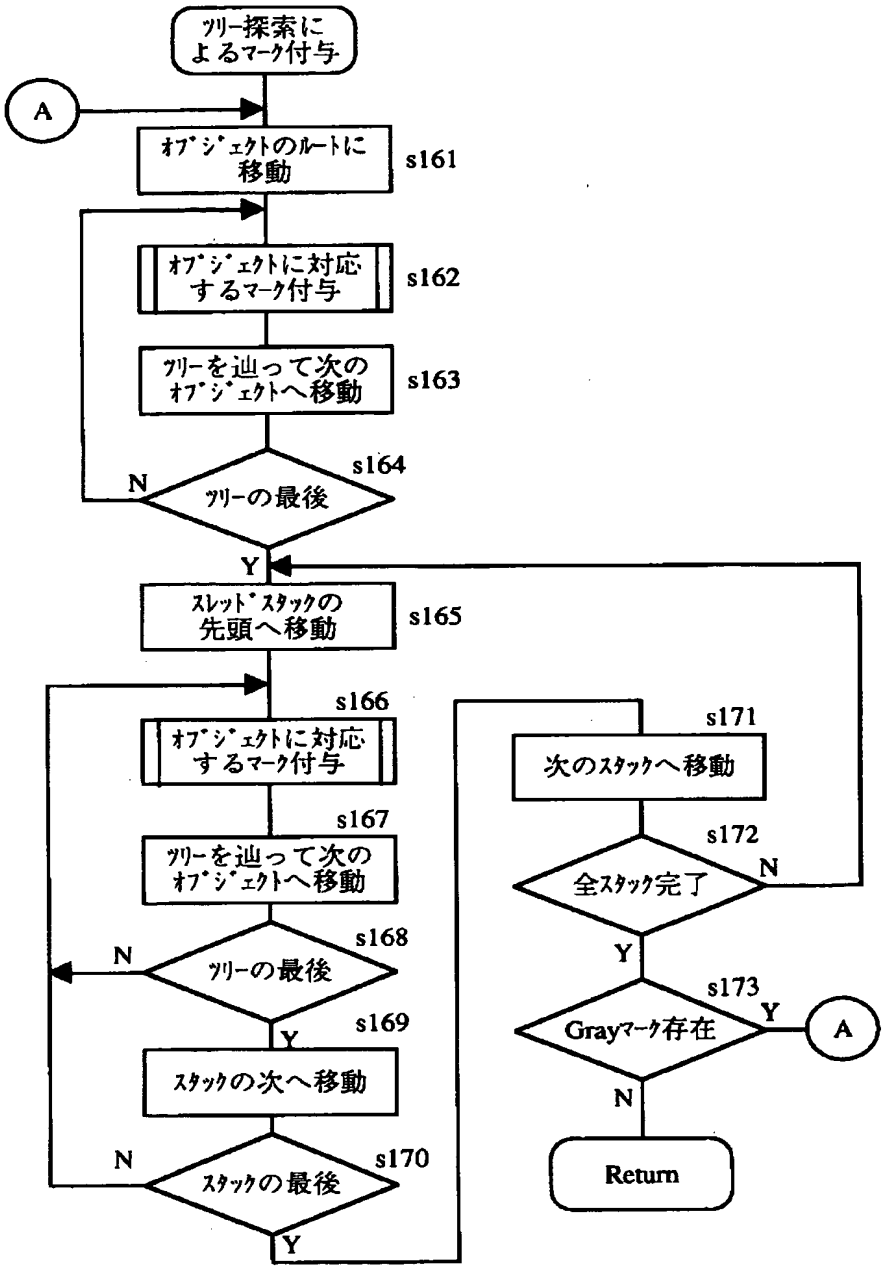
【図４０】


参照変更の
ﾏｰｸ付与
ｵﾌﾞｼﾞｪｸﾄからﾏｰｸの
位置を計算して
移動
White
N
Y
Grayにﾏｰｸ
Return

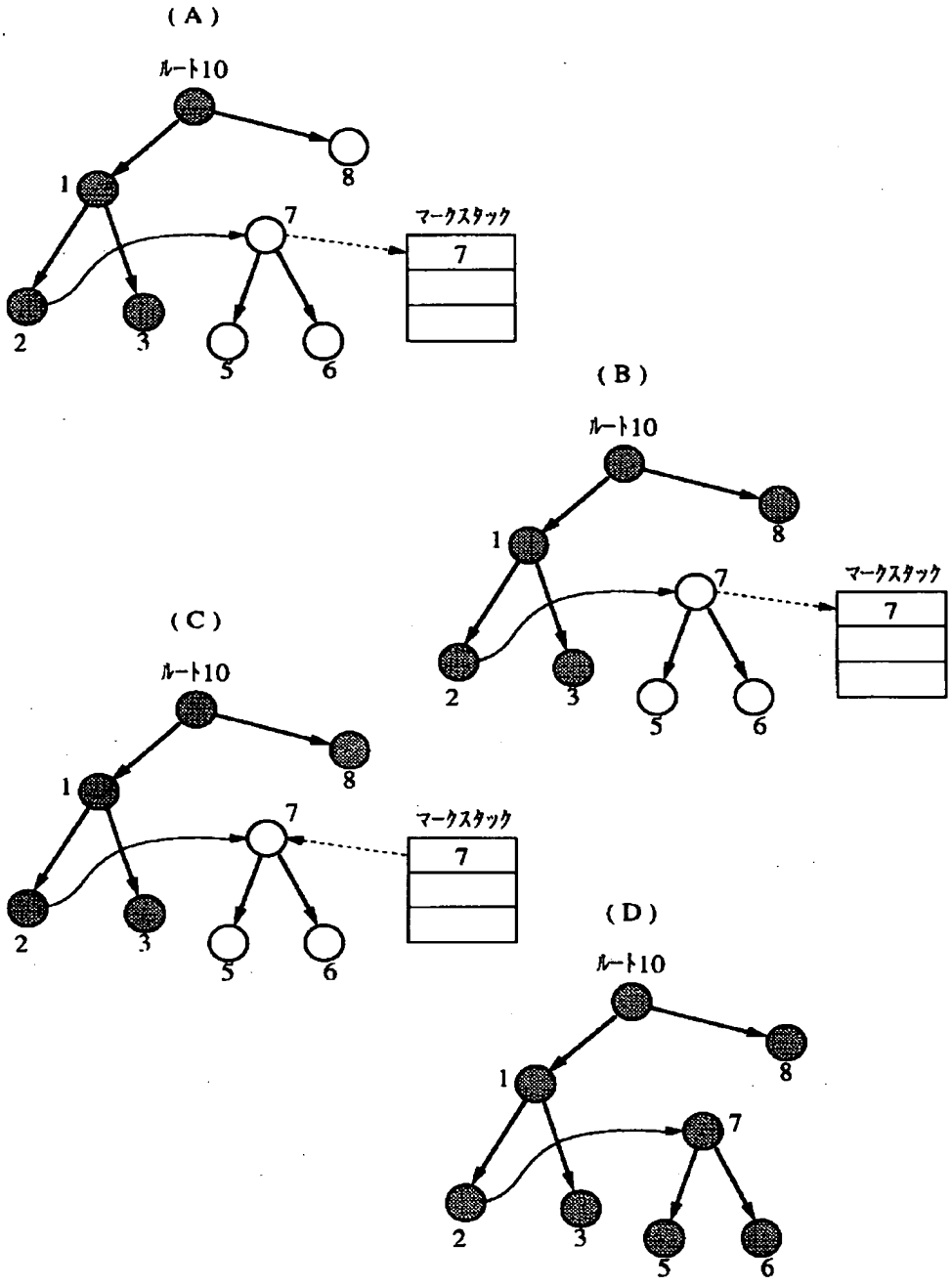
【図４２】


ｵﾌﾞｼﾞｪｸﾄに
対応するﾏｰｸ
付与
ｵﾌﾞｼﾞｪｸﾄからﾏｰｸの
位置を計算して
移動
Bkackにﾏｰｸ
Return

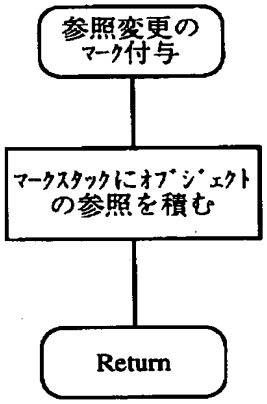
【図４１】


ツリー探索によるマーク付与
A
オブジェクトのルートに移動
s161
オブジェクトに対応するマーク付与
s162
ツリーを辿って次のオブジェクトへ移動
s163
s164
N
ツリーの最後
Y
スレッドスタックの先頭へ移動
s165
s166
オブジェクトに対応するマーク付与
s167
ツリーを辿って次のオブジェクトへ移動
s168
N
ツリーの最後
Y
s169
スタックの次へ移動
N
s170
スタックの最後
Y
s171
次のスタックへ移動
s172
全スタック完了
N
Y
s173
Grayマーク存在
Y
A
N
Return

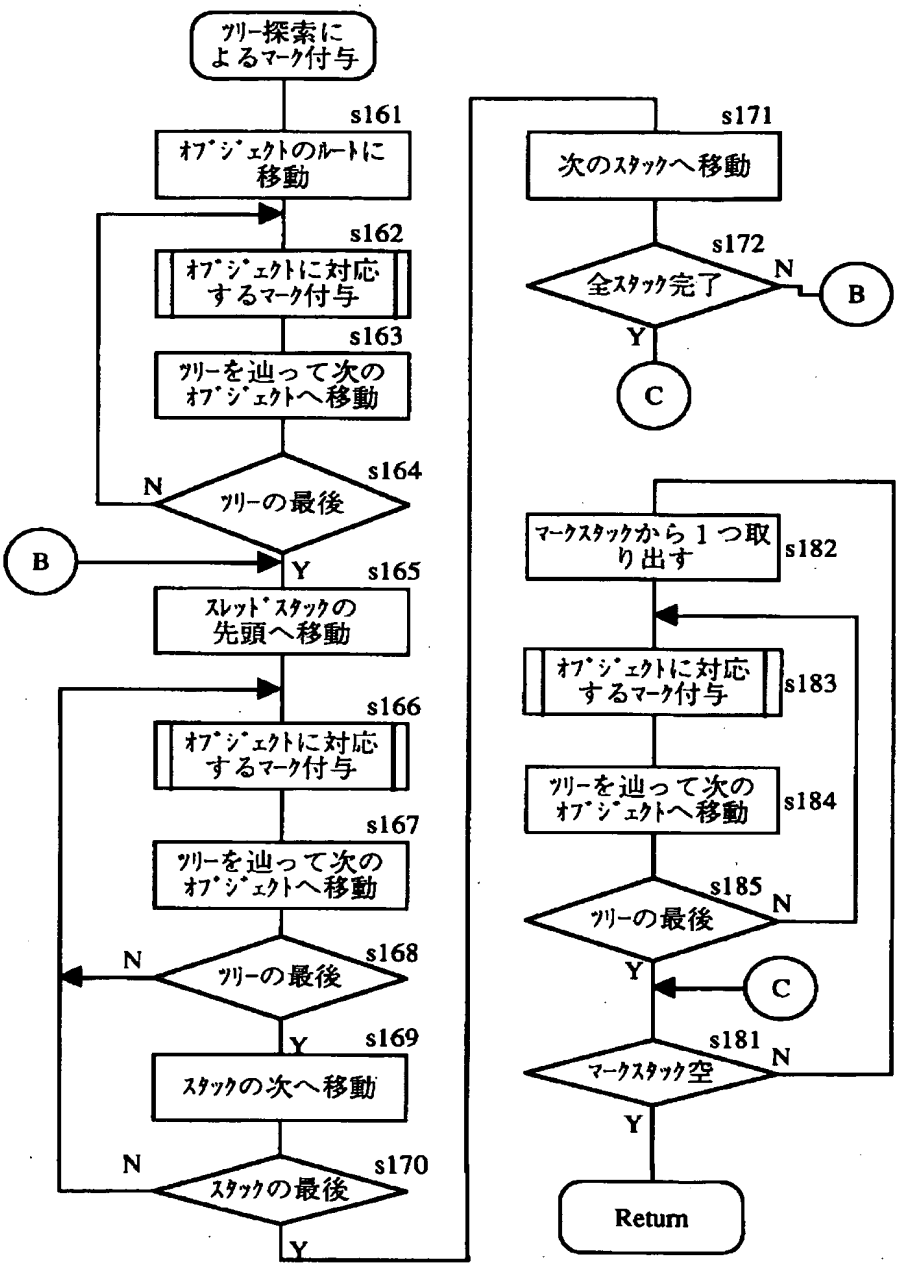
【図４３】


(A)
ルート10
1
8
7
マークスタック
7
2
3
5
6
(B)
ルート10
1
8
7
マークスタック
7
2
3
5
6
(C)
ルート10
1
8
7
マークスタック
7
2
3
5
6
(D)
ルート10
1
8
7
2
3
5
6

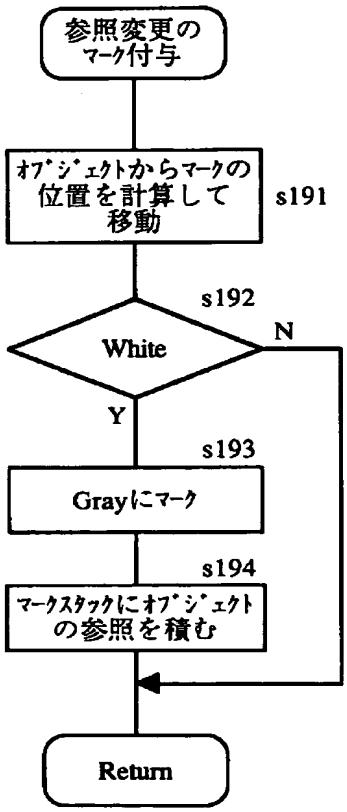
【図４４】



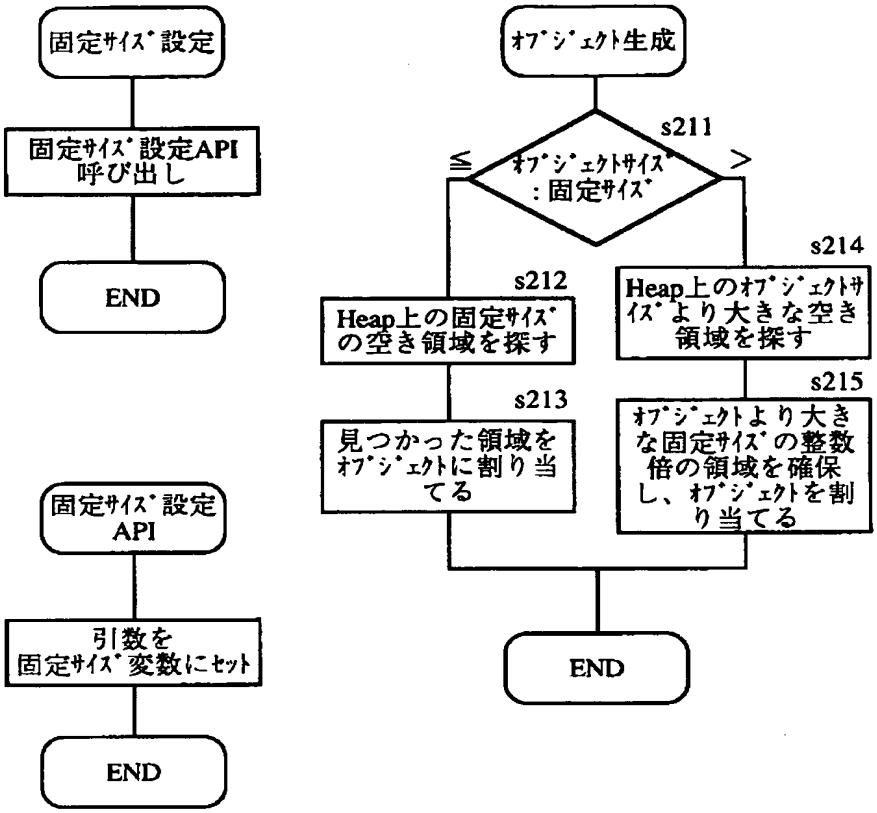
【図４５】


ツリー探索によるマーク付与
s161
オブジェクトのルートに移動
s162
オブジェクトに対応するマーク付与
s163
ツリーを辿って次のオブジェクトへ移動
s164
ツリーの最後
N
Y
B
s165
スレッドスタックの先頭へ移動
s166
オブジェクトに対応するマーク付与
s167
ツリーを辿って次のオブジェクトへ移動
s168
ツリーの最後
N
Y
s169
スタックの次へ移動
s170
スタックの最後
N
Y
s171
次のスタックへ移動
s172
全スタック完了
N
Y
B
C
マークスタックから１つ取り出す
s182
オブジェクトに対応するマーク付与
s183
ツリーを辿って次のオブジェクトへ移動
s184
s185
ツリーの最後
N
Y
C
s181
マークスタック空
N
Y
Return

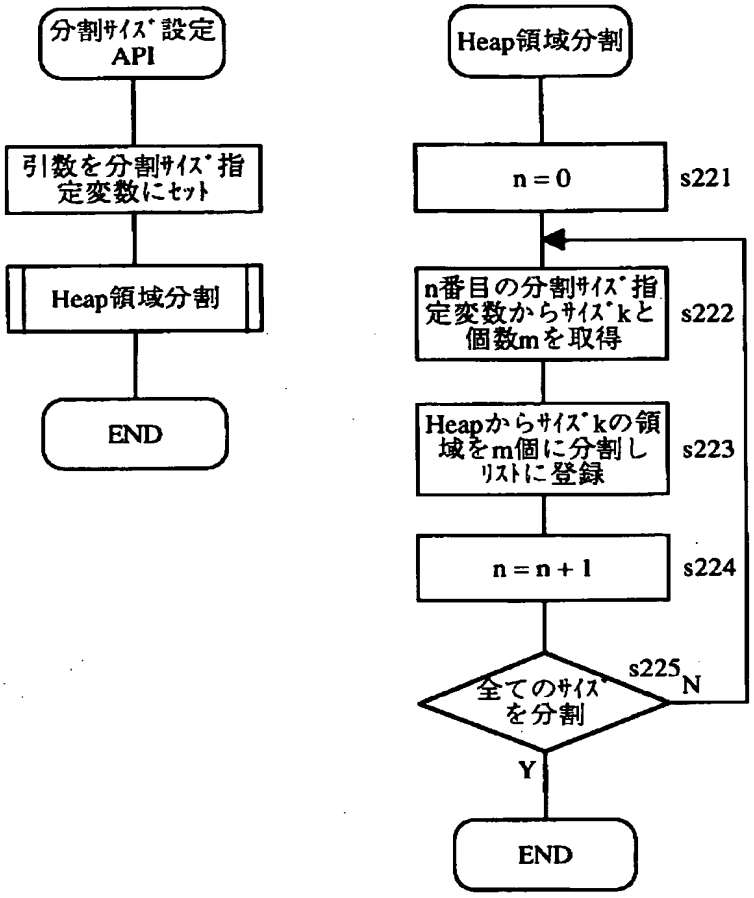
【図４７】


参照変更の
ﾏｰｸ付与
ｵﾌﾞｼﾞｪｸﾄからﾏｰｸの
位置を計算して
移動
s191
s192
White
N
Y
s193
Grayにﾏｰｸ
s194
ﾏｰｸｽﾀｯｸにｵﾌﾞｼﾞｪｸﾄ
の参照を積む
Return

【図４９】


固定ｻｲｽﾞ設定
固定ｻｲｽﾞ設定API
呼び出し
END
固定ｻｲｽﾞ設定
API
引数を
固定ｻｲｽﾞ変数にｾｯﾄ
END
ｵﾌﾞｼﾞｪｸﾄ生成
s211
≦
ｵﾌﾞｼﾞｪｸﾄｻｲｽﾞ
：固定ｻｲｽﾞ
>
s212
Heap上の固定ｻｲｽﾞ
の空き領域を探す
s213
見つかった領域を
ｵﾌﾞｼﾞｪｸﾄに割り当
てる
s214
Heap上のｵﾌﾞｼﾞｪｸﾄｻ
ｲｽﾞより大きな空き
領域を探す
s215
ｵﾌﾞｼﾞｪｸﾄより大き
な固定ｻｲｽﾞの整数
倍の領域を確保
し、ｵﾌﾞｼﾞｪｸﾄを割
り当てる
END

【図５３】



フロントページの続き

(72)発明者　栗林　博
京都府京都市右京区花園土堂町10番地
オムロン株式会社内

(56)参考文献　特開　平１－217638（ＪＰ，Ａ）
特開　平３－141442（ＪＰ，Ａ）
特開　平８－153037（ＪＰ，Ａ）
特開　平６－208502（ＪＰ，Ａ）
特開　昭61－75649（ＪＰ，Ａ）
特開　平４－38540（ＪＰ，Ａ）
発明協会公開技報・公技番号94－6618
発明協会公開技報・公技番号92－1076

(58)調査した分野(Int.Cl.', ＤＢ名)
G06F 9/46
G06F 12/00
G06F 9/44
G06F 12/02
G06F 11/14